世界著名计算机教材精选

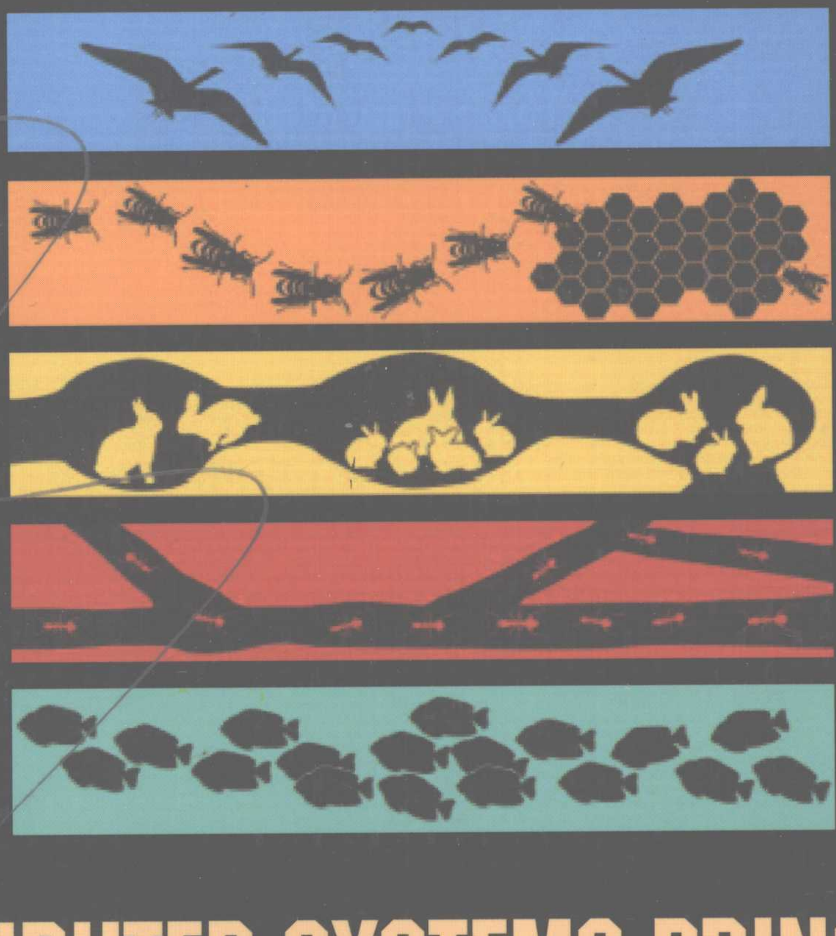
PEARSON Prentice Hall

分布式系统原理与范型

（第2版）

Andrew S. Tanenbaum
Maarten Van Steen 著

辛春生 陈宗斌 等 译



PEARSON Education

DISTRIBUTED SYSTEMS PRINCIPLES AND PARADIGMS

Second Edition

清华大学出版社

Simplified Chinese edition copyright © **2008** by **PEARSON EDUCATION ASIA LIMITED and Tsinghua University Press.**

Original English language title from Proprietor's edition of the Work.

Original English language title: Distributed Systems: Principles and Paradigms, 2E **by** Andrew S. Tanenbaum, Maarten Van Steen, Copyright © **2008**
EISBN: 0-13-239227-5
All Rights Reserved.

Published by arrangement with the original publisher, Pearson Education, Inc., publishing as Prentice Hall.
This edition is authorized for sale only in the People's Republic of China (excluding the Special Administrative Region of Hong Kong and Macao).
本书中文简体翻译版由 Pearson Education(培生教育出版集团)授权给清华大学出版社在中国境内(不包括中国香港、澳门特别行政区)出版发行。
北京市版权局著作权合同登记号 图字 01-2006-6347 号

本书封面贴有 Pearson Education(培生教育出版集团)激光防伪标签,无标签者不得销售。
版权所有,侵权必究。侵权举报电话: 010-62782989 13701121933

图书在版编目(CIP)数据

分布式系统原理与范型(第 2 版)/(美)特尼博姆(Tanenbaum, A. S.)等著;辛春生等译.
—2 版.—北京: 清华大学出版社,2008.6
(世界著名计算机教材精选)
书名原文: Distributed Systems: Principles and Paradigms, 2E
ISBN 978-7-302-17279-6

Ⅰ.分… Ⅱ.①特… ②辛… Ⅲ.分布式操作系统—教材 Ⅳ.TP316.4

中国版本图书馆 CIP 数据核字(2008)第 042078 号

责任编辑: 龙啟铭
责任校对: 徐俊伟
责任印制: 何 芊

出版发行: 清华大学出版社　　**地　　址**: 北京清华大学学研大厦 A 座
http://www.tup.com.cn　　**邮　　编**: 100084
社 总 机: 010-62770175　　**邮　　购**: 010-62786544
投稿与读者服务: 010-62776969, c-service@tup.tsinghua.edu.cn
质 量 反 馈: 010-62772015, zhiliang@tup.tsinghua.edu.cn
印 刷 者: 北京密云胶印厂
装 订 者: 北京市密云县京文制本装订厂
经　　销: 全国新华书店
开　　本: 185×260　　**印　张**: 32　　**字　　数**: 773 千字
版　　次: 2008 年 6 月第 1 版　　**印　　次**: 2008 年 6 月第 1 次印刷
印　　数: 1～3000
定　　价: 59.00 元

本书如存在文字不清、漏印、缺页、倒页、脱页等印装质量问题,请与清华大学出版社出版部联系调换。
联系电话: 010-62770177 转 3103　　产品编号: 023234-01

译 者 序

自本书的第 1 版出版以来，计算机系统技术，尤其是分布式系统技术发生了巨大的变化。根据分布式系统技术的发展和实际教学的需要，作者对本书进行了补充和更新。本书分成了两大部分。第 2～9 章讨论的是分布式系统的原理、概念和技术，包括通信、进程、命名、同步化、一致性和复制、容错性以及安全性等，而分布式应用的开发方法（即范型）在第 10～13 章中进行了讨论。但是，与前一版不同的是，本书没有在讨论范型的章节中完整地介绍每个案例研究，而是通过一个有代表性的案例来解释原理。这种介绍方法不仅精简了素材，而且可以使得读者在阅读和学习时更愉快些。全书结构清晰，内容全面经典，系统性与先进性并茂。

本书适用对象广泛，不仅可以作为学习分布式计算机的本科生和研究生的教材，对于从事分布式计算研究和工程应用的科技人员和工程技术人员来说，本书也是一本优秀的读物。

参与本书翻译、审稿的人员有辛春生、陈宗斌、周成兴、张长富、陈河南、陈红霞、张景友、易小丽、陈婷、管学岗、王新彦、金惠敏、张海峰、徐晔、戴锋、张德福、张士华、张锁玲、杜明宗、高玉琢、王涛、申川、孙玲、李振国、高德杰、宫飞、侯经国、刘淑妮、张春林、李大成、程明、张路红、张淑芝、孙先国、刘冀得、梁永翔、张广东、郁琪琳、邵长凯、蒲书箴、潘曙光、刘瑞东、李军等。

由于分布式是一种快速发展的新技术，以及译者水平有限，书中难免有错误和疏漏之处，恳请读者不吝指正。

前　言

分布式系统已经成为了计算机科学的一个快速发展的领域。从本书的前一版出版以来，分布式系统又出现了一些新的主题，如点对点计算和传感器网络，而其他主题则变得更加成熟了，比如 Web 服务和 Web 应用等。这些发展变化要求我们对这本教材进行修订。

本版对上一版进行了重大修订。我们添加了单独的一章，以反映分布式系统组织结构所取得的进展。另一个重大的修改是，本版介绍了更多的非集中式系统内容，尤其是点对点计算。我们不仅讨论了基本技术，而且还介绍了具体的应用，如文件共享、信息传播、内容传送网络和发布/订购系统等。

除了这两个重大主题外，本书还介绍了其他一些新的主题。例如，我们介绍了传感器网络、虚拟化技术、服务器集群和网格计算等。我们还特别关注了分布式系统的自我管理，随着系统不断扩展，这是一个越来越重要的主题。

当然，我们还与时俱进地给出了一些新素材。例如，在讨论一致性与复制时，我们关注的重点是更适合现代分布式系统的一致性模型，而不是最初的模型。同样，我们还添加了有关现代分布式算法的素材，包括基于 GPS 的时钟同步化和本地化算法。

与前一版相比，本版的总页码有所减少。这主要是本版弃用了诸如分布式垃圾收集和电子支付协议等主题，并对后 4 章进行了重新组织。

与前一版一样，本书分成了两大部分。第 2～9 章讨论的是分布式系统的原理，而分布式应用的开发方法（即范型）在第 10～13 章中进行了讨论。但是，与前一版不同的是，我们没有在讨论范型的章节中完整地介绍每个案例研究，而是通过一个有代表性的案例来解释原理。例如，对象激活作为一个通信原理在第 10 章关于基于对象的分布式系统中进行了介绍。这种介绍方法使得我们不仅精简了素材，而且可以使得读者在阅读和学习时更愉快些。

当然，我们仍然是努力从实践来阐述什么是分布式系统。本书介绍各种的实际系统，如 Web-Sphere MQ、DNS、GPS、Apache、CORBA、Ice、NFS、Akamai、TIB/Rendzvous、Jini 等。这些示例阐明了理论与实践的关系，使得分布式系统成为了一个令人激动的领域。

许多人以多种方式对本书作出了贡献。我们尤其要感谢 D. Rober Adams、Arno Bakker、Coskun Bayrak、Jacques Chassin de Kergommeaux、Randy Chow、Michel Chaudron、Puneet Singh Chawla、Fabio Costa、Cong Du、Deck Epema、Kevin Fenwick、Chandana Gamage、Ali Ghodsi、Giorgio Ingargiola、Mark Jelasity、Ahmed Kamel、Gregory Kapfhammer、Jeoren Ketema、Onno Kubbe、Patricia Lago、Steve MacDonald、Michel J. Bramanian、Chintan Shah、Ruud Stegers、Paul Tymann、Craig E. Wills、Reyven Yagel 和

Dakai Zhu 阅读了部分书稿，帮助修订了前一版中的一些错误，并提出了宝贵意见。

最后，我们还要感谢我们的家庭。Suzanne 已经经历过很多次这样的情况了。她从未说过“我受够了”，尽管这个念头肯定在她脑海里出现过。谢谢你！Barbara 和 Marvin 现在对教授们为谋生所做的工作有了更好的了解，并且认识到好教材和坏教材之间的差别。现在，对我来说他们是我努力创作出更多好教材的动力所在。

目　录

第 1 章　概述 …… 1

1.1　分布式系统的定义 …… 1

1.2　目标 …… 3

1.2.1　使资源可访问 …… 3

1.2.2　透明性 …… 4

1.2.3　开放性 …… 6

1.2.4　可扩展性 …… 7

1.3　分布式系统的类型 …… 12

1.3.1　分布式计算系统 …… 12

1.3.2　分布式信息系统 …… 14

1.3.3　分布式普适系统 …… 17

本章小结 …… 21

习题 …… 22

第 2 章　体系结构 …… 23

2.1　体系结构的样式 …… 23

2.2　系统体系结构 …… 25

2.2.1　集中式体系结构 …… 25

2.2.2　非集中式体系结构 …… 30

2.2.3　混合体系结构 …… 36

2.3　体系结构与中间件 …… 38

2.3.1　中断器 …… 38

2.3.2　自适应软件的常见方法 …… 40

2.3.3　讨论 …… 40

2.4　分布式系统的自我管理 …… 41

2.4.1　反馈控制模型 …… 42

2.4.2　示例：用 Astrolabe 监视系统 …… 43

2.4.3　示例：Globule 中的差分复制策略 …… 44

2.4.4　示例：Jade 的自动组件修复管理 …… 45

本章小结 …… 46

习题 …… 47

第 3 章　进程 ………… 48
3.1　线程 ………… 48
3.1.1　线程简介 ………… 49
3.1.2　分布式系统中的线程 ………… 52
3.2　虚拟化 ………… 55
3.2.1　虚拟化在分布式系统中的作用 ………… 55
3.2.2　虚拟机体系结构 ………… 56
3.3　客户 ………… 57
3.3.1　网络连接的用户接口 ………… 58
3.3.2　客户端软件与分布透明性 ………… 61
3.4　服务器 ………… 62
3.4.1　常见的设计问题 ………… 62
3.4.2　服务器集群 ………… 65
3.4.3　管理服务器集群 ………… 69
3.5　代码迁移 ………… 72
3.5.1　代码迁移方案 ………… 72
3.5.2　迁移与本地资源 ………… 75
3.5.3　异构系统中的代码迁移 ………… 77
本章小结 ………… 79
习题 ………… 80

第 4 章　通信 ………… 81
4.1　基础知识 ………… 81
4.1.1　分层协议 ………… 82
4.1.2　通信类型 ………… 87
4.2　远程过程调用 ………… 89
4.2.1　基本的 RPC 操作 ………… 89
4.2.2　参数传递 ………… 92
4.2.3　异步 RPC ………… 95
4.2.4　示例：DCE RPC ………… 96
4.3　面向消息的通信 ………… 101
4.3.1　面向消息的瞬时通信 ………… 101
4.3.2　面向消息的持久通信 ………… 104
4.3.3　示例：IBM WebSphere 消息队列系统 ………… 109
4.4　面向流的通信 ………… 113
4.4.1　为连续媒体提供支持 ………… 114
4.4.2　流与服务质量 ………… 115
4.4.3　流同步 ………… 117
4.5　多播通信 ………… 120
4.5.1　应用层多播 ………… 120

4.5.2 基于 gossip 的数据通信 …… 123
本章小结 …… 126
习题 …… 127

第 5 章 命名系统 …… 129
5.1 名称、标识符和地址 …… 129
5.2 无层次命名 …… 131
5.2.1 简单方法 …… 131
5.2.2 基于宿主位置的方法 …… 134
5.2.3 分布式散列表 …… 135
5.2.4 分层方法 …… 138
5.3 结构化命名 …… 141
5.3.1 名称空间 …… 141
5.3.2 名称解析 …… 143
5.3.3 名称空间的实现 …… 146
5.3.4 示例：域名系统 …… 151
5.4 基于属性的命名 …… 157
5.4.1 目录服务 …… 157
5.4.2 分层实现：LDAP …… 158
5.4.3 非集中式实现 …… 161
本章小结 …… 164
习题 …… 165

第 6 章 同步化 …… 167
6.1 时钟同步 …… 167
6.1.1 物理时钟 …… 168
6.1.2 全球定位系统 …… 171
6.1.3 时钟同步算法 …… 172
6.2 逻辑时钟 …… 176
6.2.1 Lamport 逻辑时钟 …… 176
6.2.2 向量时钟 …… 179
6.3 互斥 …… 182
6.3.1 概述 …… 182
6.3.2 集中式算法 …… 182
6.3.3 非集中式算法 …… 183
6.3.4 分布式算法 …… 184
6.3.5 令牌环算法 …… 186
6.3.6 四种算法的比较 …… 187
6.4 结点的全局定位 …… 188
6.5 选举算法 …… 190

6.5.1 传统的选举算法…… 190
6.5.2 无线系统环境中的选举算法…… 192
6.5.3 大型系统中的选举算法…… 193
本章小结…… 195
习题…… 196

第7章 一致性和复制…… 198
7.1 概述…… 198
7.1.1 进行复制的原因…… 198
7.1.2 作为扩展技术的复制…… 199
7.2 以数据为中心的一致性模型…… 200
7.2.1 持续一致性…… 201
7.2.2 一致的操作顺序…… 204
7.3 以客户为中心的一致性模型…… 209
7.3.1 最终一致性…… 209
7.3.2 单调读…… 211
7.3.3 单调写…… 212
7.3.4 读写一致性…… 213
7.3.5 写读一致性…… 214
7.4 复制管理…… 214
7.4.1 副本服务器的放置…… 215
7.4.2 内容复制与放置…… 216
7.4.3 内容分发…… 219
7.5 一致性协议…… 222
7.5.1 持续一致性…… 222
7.5.2 基于主备份的协议…… 223
7.5.3 复制的写协议…… 225
7.5.4 高速缓存相关性协议…… 227
7.5.5 实现以客户为中心的一致性…… 228
本章小结…… 230
习题…… 231

第8章 容错性…… 233
8.1 容错性概述…… 233
8.1.1 基本概念…… 233
8.1.2 故障模式…… 235
8.1.3 使用冗余掩盖故障…… 237
8.2 进程恢复…… 238
8.2.1 设计问题…… 238
8.2.2 故障掩盖和复制…… 240

8.2.3 故障系统的协定 …… 240
8.2.4 故障检测 …… 243
8.3 可靠的客户-服务器通信 …… 244
8.3.1 点对点通信 …… 244
8.3.2 失败时的 RPC 语义 …… 244
8.4 可靠的组通信 …… 248
8.4.1 基本的可靠多播方法 …… 249
8.4.2 可靠多播中的可扩展性 …… 249
8.4.3 原子多播 …… 252
8.5 分布式提交 …… 257
8.5.1 两阶段提交 …… 257
8.5.2 三阶段提交 …… 262
8.6 恢复 …… 263
8.6.1 概述 …… 263
8.6.2 检查点 …… 266
8.6.3 消息日志 …… 268
8.6.4 面向恢复的计算 …… 270
本章小结 …… 271
习题 …… 271

第 9 章 安全性 …… 273
9.1 安全性概述 …… 273
9.1.1 安全威胁、策略和机制 …… 273
9.1.2 设计问题 …… 277
9.1.3 加密 …… 281
9.2 安全通道 …… 287
9.2.1 身份认证 …… 287
9.2.2 消息的完整性和机密性 …… 293
9.2.3 安全组通信 …… 295
9.2.4 示例：Kerberos …… 298
9.3 访问控制 …… 299
9.3.1 访问控制中的常见问题 …… 299
9.3.2 防火墙 …… 302
9.3.3 安全的移动代码 …… 304
9.3.4 拒绝服务 …… 309
9.4 安全管理 …… 310
9.4.1 密钥管理 …… 310
9.4.2 安全组管理 …… 313
9.4.3 授权管理 …… 314
本章小结 …… 318

习题 …… 319

第 10 章 基于对象的分布式系统 …… 320
10.1 体系结构 …… 320
10.1.1 分布式对象 …… 320
10.1.2 示例：企业级 Java Bean …… 322
10.1.3 示例：Globe 分布式共享对象 …… 324
10.2 进程 …… 325
10.2.1 对象服务器 …… 325
10.2.2 示例：Ice 运行时系统 …… 327
10.3 通信 …… 329
10.3.1 把客户绑定到对象上 …… 329
10.3.2 静态远程方法调用与动态远程方法调用 …… 330
10.3.3 参数传递 …… 331
10.3.4 示例：Java RMI …… 332
10.3.5 基于对象的消息传递 …… 334
10.4 命名 …… 337
10.4.1 CORBA 对象引用 …… 337
10.4.2 Globe 对象引用 …… 338
10.5 同步 …… 340
10.6 一致性与复制 …… 341
10.6.1 入口一致性 …… 341
10.6.2 复制的调用 …… 343
10.7 容错性 …… 344
10.7.1 示例：CORBA 的容错性 …… 345
10.7.2 示例：Java 的容错性 …… 346
10.8 安全性 …… 348
10.8.1 示例：Globe …… 348
10.8.2 远程对象的安全性 …… 351
本章小结 …… 352
习题 …… 353

第 11 章 分布式文件系统 …… 354
11.1 体系结构 …… 354
11.1.1 客户-服务器体系结构 …… 354
11.1.2 基于群集的分布式文件系统 …… 358
11.1.3 对称式体系结构 …… 360
11.2 进程 …… 361
11.3 通信 …… 362
11.3.1 NFS 中的 RPC …… 362

11.3.2 RPC2 子系统 …… 363
11.3.3 Plan 9 中面向文件的通信 …… 365
11.4 命名 …… 366
11.4.1 NFS 中的命名 …… 366
11.4.2 构造全局名称空间 …… 370
11.5 同步 …… 371
11.5.1 文件共享的语义 …… 371
11.5.2 文件锁定 …… 373
11.5.3 在 Coda 中共享文件 …… 375
11.6 一致性和复制 …… 376
11.6.1 客户端缓存 …… 376
11.6.2 服务器端复制 …… 379
11.6.3 对等文件系统中的复制 …… 381
11.6.4 网格系统中的文件复制 …… 382
11.7 容错性 …… 383
11.7.1 处理 Byzantine 故障 …… 383
11.7.2 对等系统中的高度可用性 …… 385
11.8 安全性 …… 386
11.8.1 NFS 中的安全性 …… 386
11.8.2 分散式身份认证 …… 388
11.8.3 安全的对等文件共享系统 …… 391
本章小结 …… 393
习题 …… 394

第 12 章 基于 Web 的分布式系统 …… 395
12.1 体系结构 …… 395
12.1.1 传统的基于 Web 的系统 …… 396
12.1.2 Web 服务 …… 399
12.2 进程 …… 402
12.2.1 客户 …… 402
12.2.2 Apache Web 服务器 …… 403
12.2.3 Web 服务器群集 …… 405
12.3 通信 …… 406
12.3.1 超文本传输协议 …… 407
12.3.2 简单对象访问协议 …… 411
12.4 命名 …… 412
12.5 同步 …… 414
12.6 一致性与复制 …… 414
12.6.1 Web 代理缓存 …… 415
12.6.2 Web 宿主系统的复制 …… 417

12.6.3 Web应用程序的复制 …… 421
12.7 容错性 …… 423
12.8 安全性 …… 424
本章小结 …… 425
习题 …… 426

第13章 基于协作的分布式系统 …… 428
13.1 协作模型介绍 …… 428
13.2 体系结构 …… 430
13.2.1 一般方法 …… 430
13.2.2 传统的体系结构 …… 431
13.2.3 对等体系结构 …… 434
13.2.4 移动性和协作 …… 436
13.3 进程 …… 437
13.4 通信 …… 437
13.4.1 基于内容的路由 …… 437
13.4.2 支持复合订阅 …… 439
13.5 命名 …… 439
13.5.1 描述复合事件 …… 440
13.5.2 匹配事件与订阅 …… 441
13.6 同步 …… 442
13.7 一致性和复制 …… 442
13.7.1 静态方法 …… 442
13.7.2 动态复制 …… 445
13.8 容错性 …… 447
13.8.1 可靠的发布-订阅通信 …… 447
13.8.2 共享数据空间中的容错性 …… 449
13.9 安全性 …… 450
13.9.1 保密性 …… 450
13.9.2 安全的共享数据空间 …… 452
本章小结 …… 452
习题 …… 453

第14章 补充读物与参考文献 …… 454
14.1 进一步阅读的建议 …… 454
14.1.1 介绍性和综述性的著作 …… 454
14.1.2 体系结构 …… 455
14.1.3 进程 …… 455
14.1.4 通信 …… 456
14.1.5 命名 …… 456

14.1.6 同步化…… 457
14.1.7 一致性与复制…… 457
14.1.8 容错性…… 458
14.1.9 安全性…… 458
14.1.10 面向对象的分布式系统…… 459
14.1.11 分布式文件系统…… 459
14.1.12 基于 Web 的分布式系统…… 460
14.1.13 基于协作的分布式系统…… 460
14.2 参考文献…… 461

第 1 章 概述

计算机系统正在经历一场革命。1945 年是现代计算机时代的开始。从那时起一直到 1985 年，计算机一直是庞大笨重而昂贵的。即便是小型机每台也要上万美元，因此，大多数机构都只有很少的几台计算机，而且由于没有把这些计算机连接起来的手段，所以只能单独使用它们。

然而，从 20 世纪 80 年代中期开始，技术领域中两方面的进步开始使这种局面有所改观。第一项进步是高性能微处理器的开发。起初这些机器是 8 位机，但是很快 16 位、32 位乃至 64 位的 CPU 纷至沓来，并得到了普及。很多使用这些 CPU 的机器的计算性能都可以与大型机相媲美，而价格却比大型机便宜得多。

近半个世纪以来，在计算机技术改进方面所取得的成果数量之多对于其他工业来说是完全无法想象的。一台计算机的价格从 1 亿美元下降到了 1000 美元，而运算速度却从每秒 1 条指令提升到每秒 1 千万条指令，性能价格比提升了大约 1012 倍。如果汽车的性能价格比在同一时期以同样的速率提升，那么一辆“劳斯莱斯”牌汽车现在只值 1 美元，而且每消耗 1 加仑(1 加仑＝4.5461 升)汽油可以行驶 10 亿英里(1 英里＝1.6093 公里)(但是，如果果真如此，光是说明如何打开车门这一件事，可能在用户手册中就要占用长达 200 页的篇幅)。

第二项进步是高速计算机网络的发明。利用局域网技术可以将位于同一座建筑物里的数百台机器连接起来。使用这种连接方式可以在几 μs 内将少量数据从一台机器传输到另一台机器。如果要传输大量的数据，传输速率可以达到 10～1000Mb/s。利用广域网技术可以连接全球范围内数百万台机器，机器间的数据传输速率从 64Kb/s 到若干 Gb/s 不等。

现在，由于有以上这些技术可供使用，因此把若干由大量计算机组成的计算系统彼此通过高速网络相连接，不但是可行的，而且是很容易实现的。这种系统一般称为计算机网络或者分布式系统，以突出其与传统的**集中式系统**(centralized system，也称为**单处理器系统**(single processor system))的差异。传统的集中式系统一般由单个计算机及其外围设备构成，也可以包含一些远程终端。

1.1 分布式系统的定义

已经有很多文献给出了分布式系统的定义，但是不同文献给出的定义彼此不尽相同，而且没有一种令人满意。考虑到本书的目标，我们只给出一个粗略的描述：

分布式系统是若干独立计算机的集合，这些计算机对于用户来说就像是单个相关系统。

这个定义包含了两方面的内容。第一个方面是关于硬件的：机器本身是独立的。第二个方面是关于软件的：对用户来说他们就像在与单个系统打交道。这两个方面共同阐明了分布式系统的本质，缺一不可。我们先介绍关于软硬件的一些背景材料，在本章稍后再回到这两个要点上来。

与继续探讨分布式系统的定义相比，把注意力集中在分布式系统的重要特性上可能更好一些。其重要特性之一是，各种计算机之间的差别以及计算机之间的通信方式的差别对用户是隐藏的。同样，用户也看不到分布式系统的内部组织结构。另一个重要特性是，用户和应用程序无论在何时何地都能够以一种一致和统一的方式与分布式系统进行交互。

分布式系统的扩展或者升级应该是相对比较容易的。这一特性是由于下述原因：分布式系统由独立的计算机组成，但同时隐藏了其中单个计算机在系统里所承担任务的细节。即使分布式系统中某些部分可能暂时发生故障，但其整体在通常情况下总是保持可用。用户和应用程序不会察觉到哪些部分正在进行替换和维修，以及加入了哪些新的部分来为更多的用户和应用程序提供服务。

为了使种类各异的计算机和网络都呈现为单个的系统，分布式系统常常通过一个“软件层”组织起来，该“软件层”在逻辑上位于由用户和应用程序组成的高层与由操作系统组成的低层之间，如图1.1所示。这样的分布式系统有时又称为**中间件**（middleware）。

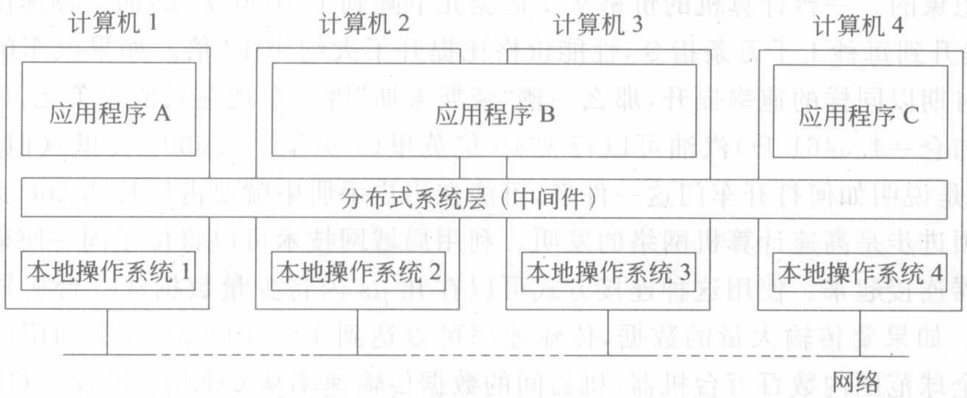


图1.1 作为中间件组织的分布式系统。中间件层延伸到了多台机器上，且为每个应用程序提供了相同的接口

现在让我们考察一下分布式系统的几个例子。第一个例子是位于一所大学或者某个公司部门里的工作站网络。该系统除了包括每个用户自己的工作站以外，还应该包括机房内的一个处理器池。这些处理器并不是分配给特定的用户，而是根据需要进行动态调配。这样的系统可能包含一个单一的文件系统，允许所有的机器通过相同的方法并且使用相同的路径名来访问所有文件。并且，当用户输入一个命令的时候，系统将寻找执行该命令的最佳位置，可能会在用户自己的工作站上直接执行该命令，也可能会在别人的一个空闲工作站上执行，还有可能由机房中某个尚未分配的处理器执行。如果系统整体外观和行为与传统的单处理器分时系统（即多用户系统）相似，那么这个系统就可以看作是一个分布式系统。

第二个例子是某个工作流信息系统，该系统支持对订单的自动处理。一般情况下，会有

来自多个不同部门的人员在不同的地点使用这样的系统。例如，销售部人员可能遍布在很大的一个区域，甚至全国。可以通过电话网络(或者蜂窝电话)连接到系统的膝上型计算机来下达订单。收到的订单由系统自动传送到计划部，接着新的内部调货订单就会送达仓储部，同时由财务部处理账单。该系统自动将订单传送到相关人员手中。用户根本看不到系统中订单处理的物理流程；对于用户来说，这些订单像是由一个集中式数据库处理的一样。

最后一个例子是万维网。它提供了一种简单、一致并且统一的分布式文档模型。要查看某个文档，用户只需激活一个引用(即链接)，文档就会显示在屏幕上。理论上(但是目前在实际中并不是这样)并不需要知道该文档来自于哪个服务器，更用不着关心服务器所在的位置。要发布一个文档也很简单：只需要赋给它一个唯一的 URL 名，让该 URL 指向包含文档内容的本地文件即可。如果万维网向用户呈现的是一个庞大的集中式文档系统，也可以认为它是一个分布式系统。但是，我们现在还无法做到这一点。例如，用户明白这样一个事实：文档位于不同的地点，并由不同的服务器处理。

1.2 目 标

如果仅仅是因为有能力组建分布式系统就去组建它，这并不是明智之举。毕竟，以现在的技术，可以在一台个人计算机上安装 4 个软驱。但这样做毫无意义。在本节中，我们将讨论要使得一个分布式系统真正物有所值必须具有的 4 个关键目标，即分布式系统必须能够让用户方便地访问资源；必须隐藏资源在一个网络上分布这样一个事实；必须是开放的；必须是可扩展的。

1.2.1 使资源可访问

分布式系统的最主要目标是使用户能够方便地访问远程资源，并且以一种受控的方式与其他用户共享这些资源。资源几乎可以是任何东西，其典型例子包括打印机、计算机、存储设备、数据、文件、Web 页以及网络，但这些只是资源中的一小部分。有很多理由要求资源共享，一个显而易见的理由是降低经济成本。例如，让若干用户共享一台打印机比为每一位用户购买并维护一台打印机要划算得多。基于同样原因，共享超级计算机和高性能存储系统这样昂贵资源也是很有意义的。

用户连接到资源以后，协作和信息交换也会变得更加方便，在这方面因特网是一个最好的例子。它通过采用交换文件、邮件、文档、音频以及视频等数据的简单协议而获得了成功。因特网的连接能力造就了众多的虚拟团体，这些团体由地域上分散的人们组成，他们通过**群件**(groupware)协同工作。群件是支持协作编辑、远程会议等活动的软件。因特网的连接能力还使电子商务成为可能，现在通过电子商务我们不用去商店就可以买卖各种货物。

然而，在连接能力和共享功能不断加强的同时，安全性也变得愈发重要。就当前的实际情况而言，系统在通信窃听或入侵方面进行的防范是极为有限的。密码以及其他敏感信息常以明文(未加密)形式通过网络发送或者存储在服务器上，我们只能期望这些服务器是可信任的。从这个意义上来说，还有许多需要改进的地方。例如，现在只需要提供信用卡号码

就可以订购货物，极少能对顾客是否确实拥有该信用卡进行验证。将来，只有在使用读卡机证实顾客确实拥有该信用卡实物的情况下，顾客才能下订单。

另外一个安全方面的问题是对通信的跟踪行为，其目的是建立一个针对特定用户的档案以记录该用户的偏好(Wang 等 1998)。这样的跟踪行为显然侵犯了隐私权，特别是在用户不知情的情况下更是如此。相关的问题是日益增长的连接能力会导致发生不希望的通信，比如垃圾电子邮件。在这些情况下，我们需要利用专门的信息过滤器来保护自己，这些过滤器可以根据传入信息的内容来决定允许哪些信息通过。

1.2.2 透明性

分布式系统的重要目标之一是将它的进程和资源实际上在多台计算机上分布这样一个事实隐藏起来。如果一个分布式系统能够在用户和应用程序面前呈现为单个计算机系统，这样的分布式系统就称为是**透明的**(transparent)。我们首先研究一下分布式系统中存在哪些种类的透明性，然后再分析是否在任何情况下都必须保证透明性。

1.2.2.1 透明的类型

透明性的概念可以运用到分布式系统中的各个方面，如图 1.2 所示。

透明性	说明
访问	隐藏数据表示形式的不同以及资源访问方式的不同
位置	隐藏资源所在位置
迁移	隐藏资源是否移动到另一个位置
重定位	隐藏资源是否在使用过程中移动到另一个位置
复制	隐藏是否对资源进行复制
并发	隐藏资源是否由若干相互竞争的用户共享
故障	隐藏资源的故障和恢复

图 1.2 分布式系统中不同形式的透明性(ISO 1995)

访问透明性(access transparency)是指对不同数据表示形式以及资源访问方式的隐藏。例如，要将一个整型数从基于 Intel 处理器的工作站传输到一台 Sun SPARC 机器上，就必须考虑到 Intel 处理器传输数据字节的顺序是 little endian 格式(先传输数据的高位字节)，而 SPARC 处理器使用的是 big endian 格式(先传输数据的低位字节)。在数据表示形式上还会存在其他差别。例如，分布式系统中的多个计算机系统运行的可能是不同的操作系统，这些操作系统的文件命名方式不同。文件命名方式的差异以及由此引发的文件操作方式的差异都应该对用户和应用程序隐藏起来。

透明性类型中很重要的一类是关于资源位置的。**位置透明性**(location transparency)指的是用户无法判别资源在系统中的物理位置。命名在确保位置透明性方面扮演了重要角色。特别是，位置透明性可以通过仅为资源分配逻辑名来取得，这种逻辑名和资源的位置完全无关。举一个这种名字的例子，从 http://www.prenhall.com/index.html 这个 URL 看不出 Prentice Hall 的主 Web 服务器的所在位置，同样也看不出 index.html 是一直位于现在所在的位置还是只是最近才移到那里的。如果分布式系统中的资源移动不会影响该资源

的访问方式，就可以说这种分布式系统就能提供**迁移透明性**(migration transparency)。如果资源可以在接受访问的同时进行重新定位，而不引起用户和应用程序的注意，拥有这种资源的系统无疑会更加强壮，也就可以说这种系统能提供**重定位透明性**(relocation transparency)。重定位透明性的一个例子是，当移动通信用户从一个地点移动到另一个地点时，可以一直使用他们的无线膝上型计算机，甚至连暂时的中断连接都不需要。

就像我们即将介绍的那样，复制在分布式系统里扮演了重要角色。例如，如果要增加系统的可用性或者提升系统性能，可以对资源进行复制，然后把其副本放置在访问该资源的位置附近。**复制透明性**(replication transparency)是指对同一个资源存在多个副本这样一个事实的隐藏。为了对用户隐藏复制行为，所有副本的名字都必须相同。这样，支持复制透明性的系统必须同时也支持位置透明性，不然系统就无法引用位于不同位置的多个副本。

我们已经提到过，分布式系统的重要目标之一是实现资源共享。多数情况下，资源共享是通过协作方式完成的，就像在通信中那样。然而，实际中还有许多竞争式资源共享的例子。例如，两个独立用户将各自的文件存储在同一台文件服务器上，或者访问位于一个共享数据库中的同一批表。在这种情况下，重要的是让任何一个用户都不会感觉到他人也在使用自己正使用的资源。这种现象称为**并发透明性**(concurrency transparency)。必须确保对共享资源的并发访问不会破坏资源的一致状态。一致性可以通过使用锁定机制来保证。通过这种机制，用户轮流对请求的资源进行独占访问。更加精致的技术是运用事务处理机制，然而就像在后续章节中将会介绍的那样，在分布式系统中要实现事务处理机制是很困难的。

关于分布式系统还有一个流行的定义，是由 Leslie Lamport 提出的："如果你从未听说过的某台计算机的崩溃导致你什么也干不了了，就说明你所打交道的是一个分布式系统"。这个描述道出了分布式系统设计上的另一个重要问题：故障处理机制。如果一个分布式系统提供**故障透明性**(failure transparency)，就意味着用户不会注意到某个资源(也许他从未听说过这个资源)无法正常工作，以及系统随后从故障中恢复的过程。屏蔽故障是分布式系统中最困难的问题之一。如果我们提出某些接近现实的假设，那么这种屏蔽甚至是不可能的，我们将在第 7 章中讨论这个问题。在故障屏蔽方面最大的困难在于无法区别出现故障的资源和反应速度极慢的资源。例如，在连接一台繁忙的 Web 服务器时，浏览器将会超时并且报告该 Web 页不可用。在这种情况下，用户无法判定该服务器是否真正崩溃了。

与分布式系统相关的透明性的最后一种类型是持久性透明性，它指的是对资源位于易失性存储器中还是位于磁盘上的隐藏。例如，许多面向对象的数据库提供直接调用存储对象的方法的功能。在这个调用过程中发生的事情是，数据库服务器首先将对象的状态数据由磁盘复制到主存储器中再执行操作，然后可能会把状态写回到辅助存储装置中去。而用户觉察不到服务器将状态数据在主存储器和辅助存储器之间进行了移动。持久性在分布式系统中扮演着重要角色，而且它对于非分布式系统也是同样重要的。

1.2.2.2 透明度

虽然对于任何分布式系统来说，实现分布透明性通常是可取的，但是在某些情况下，把所有分布情况都对用户屏蔽得严严实实并不见得有好处。这样的一个例子是，如果您订阅了一份电子报刊，并且要求它在每天本地时间早上 7 点之前就寄到邮箱里，而现在您处在地球另一端的一个不同时区，您就会发现"早报"不再是早上寄到的了。

同样，一个将分别位于旧金山和阿姆斯特丹的两个进程连接起来的广域分布式系统不可能在35ms内将某个消息从其中一个进程发送到另一个进程，这是由自然规律（光信号或者电信号传输速度的限制）决定的。实践表明，通过计算机网络发送这样的消息实际上需要数百ms。信号传输的速度不仅受光速限制，而且还要受到线路中交换机处理能力的限制。

另外，还必须在高度的透明性和系统性能之间进行权衡，得出折衷方案。例如，许多因特网应用程序会不断尝试连接到某台服务器，多次失败后才最终放弃。这种在用户转向另一台服务器之前竭力隐藏服务器短暂故障的企图会导致整个系统变慢。在这种情况下，最好是让用户早一些放弃，或者至少允许用户取消这种连接的尝试。

另一个例子是，我们必须使位于各大洲的某个资源的多个副本在任何时候都保持一致。换句话说，如果对其中一个副本进行了改动，在进行其他任何操作之前必须先将这个改动传播给所有其他副本。很明显，这时即使是一个简单的更新操作也可能需要花费数秒钟才能完成，这是不可能对用户隐藏的。

结论是，在设计并实现分布式系统时，把实现分布的透明性作为目标是正确的，但是应该将它和其他方面的问题（比如性能）结合起来考虑。

1.2.3 开放性

分布式系统还有一个重要目标，即开放性。一个**开放的分布式系统**（open distributed system）就是这样的系统：它根据一系列准则来提供服务，这些准则描述了所提供服务的语法和语义。例如，在计算机网络中，其准则规定了发送和接收的消息的格式、内容及含义。对这些准则进行形式化，就产生了协议。在分布式系统中，服务通常是通过**接口**（interface）指定的，而接口一般是通过**接口定义语言**（interface definition language，IDL）来描述的。用IDL编写的接口定义基本上只是记录服务的语法，即这些接口定义明确指定可用函数的名称、参数类型、返回值，以及指出可能出现的异常等。比较困难的是确切地说明这些服务所要完成的工作，也就是接口的语义。在实际应用中，这样的接口说明一般是用一种非正式的方式，通过自然语言给出的。

说明良好的接口定义允许需要某个接口的任意进程与提供该接口的进程进行通信。同时它也允许独立的双方各自完成截然不同的接口实现，这样就会出现运行方式完全相同的两个独立的分布式系统。良好的接口规范说明是完整且中立的。"完整"意味着完成接口实现不可少的所有内容都已经规定好了。然而，有许多接口定义都是不完整的，这样就需要开发人员添加针对特定实现的细节。同样重要的是，接口规范说明不应描述接口的实现应该是什么样子，它应该是中立的。完整性和中立性对于实现互操作性和可移植性很重要（Blair和Stefani 1998）。**互操作性**（interoperability）刻画了来自不同厂商的系统或组件的两种实现能够在何种程度上共存并且协同工作，这种共存和协同工作只能依赖于通过双方在公共标准中规定的各自所提供的服务来完成。**可移植性**（portability）刻画了这样的性能：如果为分布式系统A开发了某个应用程序，并且另一个分布式系统B与A具有相同的接口，该应用程序在不做任何修改的情况下在B上执行的可行程度。

开放的分布式系统的另一个重要目标是，它必须是灵活的，要能够方便地把由不同开发人员开发的不同组件组合成整个系统。同时，还必须能够方便地添加新组件、替换现有的组

件，而不会对那些无须改动的组件造成影响。换句话说，开放的分布式系统应该是**可扩展的**(extensible)。例如，在灵活的系统中，要添加可运行于另一个操作系统上的部件应该是比较容易的，即使要更换整个文件系统也不应该太困难。但根据我们从日常实践中得到的经验，要实现灵活性是比较困难的。

将策略与机制分离

要在开放的分布式系统中获得灵活性，关键是要把系统组织成规模相对较小而且容易替换或修改的组件的集合。这意味着我们应该不只是提供最高层的接口定义(也就是对用户和应用程序可见的那部分内容)，还应该提供系统内部各部分间的接口定义，并且描述它们的交互方式。这种方案是比较新颖的。许多比较老的系统，甚至包括某些现代的系统，都是使用一种单一的形式构建起来的。在这种较老的方案中，逻辑上组件是彼此分离的，但是作为一个整体加以实现，成为一个庞大的程序。在按这种方案实现的系统中，要在不影响整个系统的前提下更换或者修改某个组件是很困难的。单一系统并不是开放的，而是趋向于封闭的。

在分布式系统中，进行改动的需求常常出自于某个不能对特定用户或者应用程序提供最优策略的组件。举一个例子，考虑一下万维网中的缓存机制。浏览器一般允许用户修改缓存策略，例如，可以指定缓存的大小，以及指定不断检查缓存的文档以确保与真正的文档一致，还是每次会话只检查一次。然而，用户无法操纵缓存的其他参数，比如一个文档应该在缓存中保持多久，以及缓存空间填满时应该删除哪个文档。同时，也无法根据文档内容来决定是否对其进行缓存。例如，用户可能想要缓存铁路时刻表，因为他知道时刻表一般极少改动，但他不会缓存当前高速公路上的交通状况信息。

我们需要的是将策略与机制相分离。在以上 Web 缓存的例子中，理想情况下浏览器应该仅提供存储文档的功能，同时允许用户决定每个文档是否需要保存以及保存多久。在实际应用中，这可以通过为用户提供大量可以动态设置的参数来实现。更好的做法是用户可以通过某个能插入到浏览器中的组件来定制实现自己的策略。当然，这种组件必须具有浏览器可以识别的接口，以允许浏览器调用与该接口相对应的过程。

1.2.4 可扩展性

通过互联网，世界范围的互连正在变得像给世界任何地方的人寄张明信片一样平常。考虑到这种情况，分布式系统开发者都将可扩展性作为最重要的设计目标之一。

系统的可扩展性至少可以通过三个方面来度量(Neuman 1994)。首先，系统要能在规模上扩展，也就是说可以方便地把更多的用户和资源加入到系统中去。其次，如果系统中的用户和资源可以相隔极为遥远，这种系统就称为地域上可扩展的系统。最后，系统在管理上是可扩展的，也就是说，即使该分布式系统跨越多个独立的管理机构，仍然可以方便地对其进行管理。但是，在以上这三个方面或其中某一两个方面上可扩展的系统常常在扩展之后表现出性能的下降。

1.2.4.1 可扩展性问题

当一个系统需要进行扩展时，必须解决多方面的问题。首先考虑规模上的扩展。在需要支持更多的用户或资源时，我们常常受到集中的服务、数据以及算法所造成的限制（如图1.3所示）。例如，许多服务是以集中的方式实现的，它们由分布式系统中一台特定的计算机上运行的单个服务器来提供。这种方案存在的问题是显而易见的：用户增多时该服务器将成为系统的瓶颈。即使它拥有无限的处理能力和存储能力，在系统达到一定规模后与该服务器的通信也将发生困难，从而使得系统规模无法继续增长。

概念	实例
集中式服务	供所有用户访问的单个服务器
集中式数据	单个在线电话簿
集中式算法	根据完整的信息来安排路由

图 1.3 可扩展性限制的实例

但是，有时不可避免地要使用单个服务器。设想这样一种情况，有一项服务用来管理高度机密的信息，例如医疗记录、银行账户、个人贷款等。这时最好把这项服务放在高度安全的一台服务器上，并且通过专门的网络组件阻止分布式系统的其他部分对其进行访问。把服务器复制到多个位置以提高性能的做法是不可取的，因为这样会使得该服务更容易受安全攻击。

集中式数据与集中式服务一样有缺陷。如果我们要保存5000万人的电话号码和地址，会出现什么情况？假设每条数据记录占50个字符，一个2.5GB的硬盘就可以提供足够的存储空间。但是这里有一个同样的问题，如果只用一个数据库，无疑会使得这个数据库的进出通信线路中都充满了数据。同样，设想一下如果因特网的**域名系统**（domain name system，DNS）只用单个表来实现，会出现什么情况？DNS维护世界范围内数百万台计算机的信息，提供用于定位Web服务器的关键服务。如果每个解析URL的请求都必须传送到唯一的DNS服务器，那么显然不会再有人使用Web了（当然，如果真是这样的话，DNS服务器的拥挤问题也许就会解决了）。

最后，集中式算法也有缺陷。在大型分布式系统中，海量的信息必须在许多线路之间进行路由传送。从理论上来说，完成这种传送工作最好的办法是收集关于所有计算机及线路负载的完整信息，然后利用图论的算法来计算最优的路由线路，计算结果随即应用到系统中以改善信息路由的性能。

但是，收集并传送所有的输入和输出信息的做法本身就有弊端，因为这些消息会造成部分网络过载。实际上，如果任何一个算法是通过从所有站点收集信息，然后将这些信息发送到某个计算机加以处理，随后又向系统发布计算结果的方式来执行的，那么就不应该采用这种算法，只应该采用分散式算法。分散式算法一般有下列与集中式算法不同的特性：

(1) 没有任何计算机拥有关于系统状态的完整信息。

(2) 计算机只根据本地信息做出决策。

(3) 某台计算机的故障不会使算法崩溃。

(4) 没有这样的假设：存在全局性的时钟。

前三个特性来自于我们前面已经强调过的内容，最后一个特性的重要性也许不那么明显，但它也同样重要。如果某个算法以这样的语句开头："在 12：00：00 所有计算机必须记录其输出队列的大小"，这样的算法必定失败，因为无法使各个计算机的时钟完全同步。算法必须考虑到时钟无法完全同步的问题。系统越大，这种不确定度就越大。在一个小型局域网内，通过相当的努力可以把所有时钟的同步误差缩小到几个 ms，但是要在全国范围甚至全球范围的网络里做到这一点是极其困难的。

地域上的可扩展性也存在着问题。难以对现有的为局域网设计的分布式系统进行扩展的主要原因之一是这些系统基于同步通信。在这种通信方式下，请求服务的一方(一般称为客户)在得到回应以前是处于阻塞状态的。这种方法通常适用于局域网，因为局域网内两台机器之间的通信最多只需要数百 μs。然而，在广域系统中，必须考虑到进程间通信可能需要花费数百 ms，比局域网慢三个数量级。如果要在广域系统中使用同步通信建立交互式应用程序，应非常小心，还需要极大的耐心。

另一个困扰地域可扩展性的问题是，广域网中的通信从本质上说是不可靠的，而且几乎总是点对点的。相比之下，局域网一般提供高度可靠的基于广播的通信功能，使得在局域网中开发分布式系统要相对容易得多。例如，服务定位的问题。在局域系统中，进程可以简单地向每台计算机广播一条消息，询问每台计算机是否正运行该进程所需的服务。只有那些提供该服务的机器才会做出响应，在应答消息中包含该机器的网络地址。这种定位方案在广域系统中是不可想象的。在广域系统中必须设计专门的定位服务，这种服务必须能够扩展到世界范围来使用，并且能够向数十亿用户提供服务。在第 4 章中我们将详细介绍这种服务。

地域扩展与困扰规模扩展的集中式解决方案中存在的问题也息息相关。如果一个系统使用了许多集中式组件，显然，地域扩展性将会由于广域通信造成的性能和可靠性上的问题而受到制约。另外，集中式组件还会导致网络资源的浪费。设想一下整个地区只使用一台邮件服务器会发生什么情况，这意味着如果要向你的邻居发送一封电子邮件，必须先连接到中心邮件服务器，它可能距离你的位置数百英里之遥。很明显，不能采取这种方法。

最后，有一个难以解决的问题，它在许多情况下还没有答案，这就是如何对一个跨越多个独立管理域的分布式系统进行扩展。要解决的关键问题是在资源使用(以及付费)、管理和安全问题上各域有着相互冲突的策略。

例如，一个位于单个域中的分布式系统的许多组件通常可以得到在同一个域中工作的用户的信任。在这种情况下，系统管理部门可能已经对应用程序进行过测试和认证，并且采取了专门的措施以确保这些组件不会受到破坏。从本质上说，用户信任的是系统管理员。然而，这种信任无法自动拓展到域的边界之外。

如果分布式系统拓展到了另一个域，必须采取两种类型的安全措施。首先，分布式系统必须保护自己免受来自新域的恶意攻击。例如，来自新域的用户可能只应该拥有对于系统原始域中文件服务的读权限。同样的道理，像昂贵的图像排版设备或者高性能计算机这样的设备不应该允许其他用户使用。其次，新域也必须保护自身免受来自分布式系统的恶意攻击。典型的例子是下载类似 Web 浏览器中的 Java 小程序时要小心。基本上，新域无法预料这样的外来代码会导致什么后果，因而可能会严格限制对这种代码的访问权限。问题是如何实施这种限制，对此我们将在第 9 章中加以讨论。

1.2.4.2 扩展技术

我们已经讨论了一些关于可扩展性的问题，现在面对的问题是在一般情况下如何解决这些问题。由于分布式系统中的可扩展性问题表现为服务器和网络能力有限所造成的性能问题，因此基本上只有三种扩展技术：隐藏通信等待时间、分布技术以及复制技术（Neuman 1994）。

隐藏通信等待时间的技术对于地域扩展是比较适用的。基本的想法很简单：尽量避免等待远程服务对请求的响应。例如，当对远程计算机的某个服务发出请求时，在发出请求端，除了等待服务器响应之外，还可以利用这段时间做其他工作。本质上，以这种方式构建的应用程序使用的是**异步通信**（asynchronous communication）。当响应到来时，应用程序产生中断，并且调用专门的处理程序对前面发出的请求进行处理。异步通信通常用于批处理系统以及并行应用程序中，这些系统或程序或多或少都拥有在某项任务等待通信完成时调度其他独立任务执行的能力。另外一种选择是启动一个新的控制线程来执行请求，虽然该控制线程因为等待响应而受到阻塞，但是进程中的其他线程可以继续执行。

然而，有许多应用程序无法充分利用异步通信的优点。例如，在交互式应用程序中，如果用户发送了一个请求，他一般只会无所事事地等待着回答。在这种情况下，较好地解决方案是减少通信的总量，例如，把通常由服务器完成的部分计算工作交由请求服务的客户端进程来进行。在使用表单来访问数据库的情况下，这种方案是行之有效的。通常，填写表单时需要为每个字段发送一条单独消息，然后等待服务器的响应，如图 1.4(a)所示。例如，服务器可能在接受某个项目之前先检查该条目是否存在句法错误。更好的方案是先把用于填写表单（可能还有检查条目）的代码传输给客户，然后由客户返回完整的表单，如图 1.4(b)所示。这种传输代码的解决方案目前在 Web 中通过使用 Java 小程序的形式得到了广泛支持。

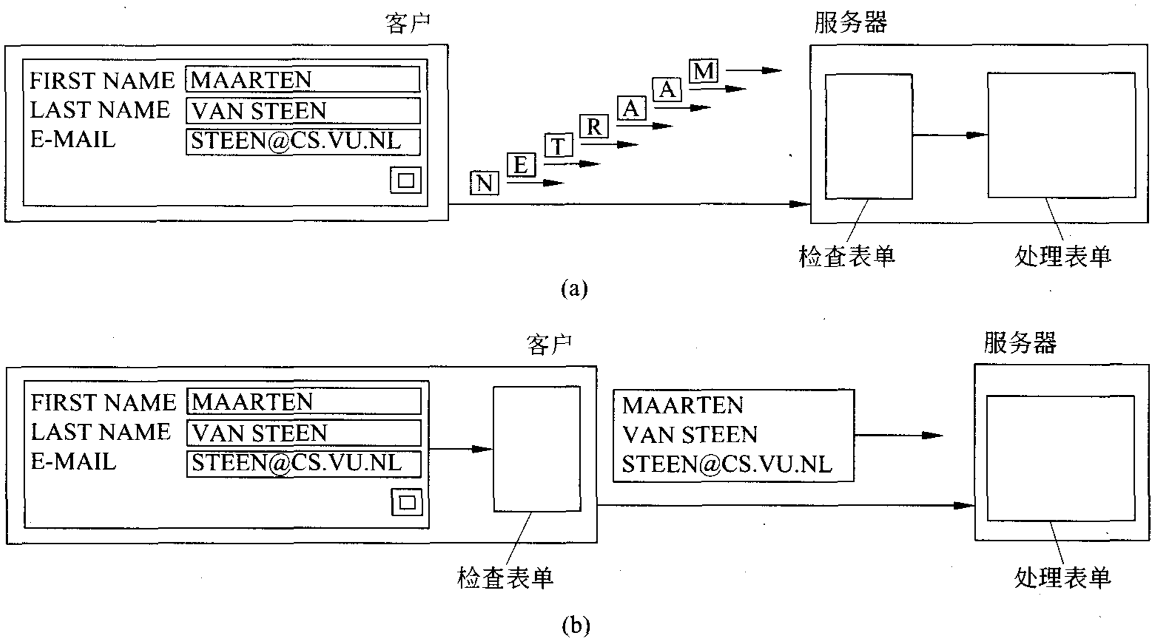


图 1.4 两种方案的区别

(a) 由服务器检查正在填写的表单；(b) 由客户检查正在填写的表单

另一项重要的扩展技术是**分布**(distribution)技术。分布技术把某个组件分割成多个部分,然后再将它们分散到系统中去。使用分布技术的一个例子是因特网 DNS。DNS 名字空间是由**域**(domain)组成的分层树状结构,域又划分为互不重叠的**区**(zone),如图 1.5 所示。每个区内的名字都由单个域名服务器处理。在不涉及太多细节的情况下,可以把每个路径名想象成因特网上的主机名,与该主机的网络地址相关。从根本上来说,解析一个名字意味着返回与该名字相关联的主机的网络地址。例如,要解析名字 nl. vu. cs. flits,首先要把它发送给 Z1 区的服务器(如图 1.5 所示),该服务器会返回 Z2 区的服务器地址,该名字的其余部分 vu. cs. flits 会发送给 Z2 区的服务器。而 Z2 区服务器将返回 Z3 区服务器的地址,Z3 区服务器将处理名字的最后一部分并且返回相应的主机地址。

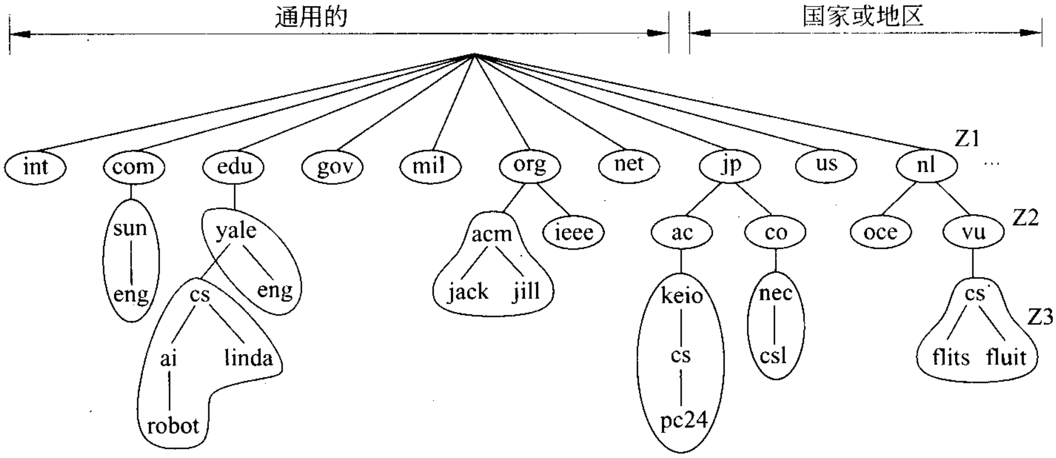


图 1.5 将 DNS 名字空间划分为区的例子

以上这个例子阐明了由 DNS 提供的命名服务是如何分布到多台计算机上进行的,这种做法避免了单个服务器不得不处理所有名字解析请求所面临的困境。

再来看看另一个例子:万维网(WWW)。对于大多数用户来说,Web 呈现给他们的是一个庞大的基于文档的信息系统,其中的每个文档都拥有以 URL 形式给出的唯一名字。从概念上讲,Web 看起来就像只有一个单独的服务器一样。然而,Web 在物理上是分布到非常多的服务器上的,每台服务器处理一部分 Web 文档。通过编码,把处理文档的服务器的名字变成该文档的 URL 中的一部分。正是因为对文档进行了分布处理,Web 才得以扩展到现在的规模。

考虑到可扩展性的问题通常表现为性能的下降,对组件进行复制并将副本分布到系统各处通常是一个好办法。复制不仅能够增加可用性,而且还有助于组件间的负载平衡,从而使性能得到提高。同样,对于在地域上比较分散的系统来说,在请求者附近有一份副本可以在很大程度上隐藏前面提到的通信等待时间的问题。

缓存(caching)是复制的一种特殊形式,二者之间的区分通常很难划定。与复制相同,缓存一般是在访问资源的客户附近制作该资源的副本。然而,与复制不同的是,是否进行缓存是由要访问资源的客户决定的,而不是由资源拥有者决定的。

缓存和复制都存在严重的缺点,这些缺点可能会对可扩展性造成不良影响。进行缓存或者复制以后,由于资源存在多个副本,修改其中的一个会导致它与其他副本不相同,从而导致**一致性**(consistency)方面的问题。

可以在多大程度上容忍不一致性主要取决于资源的用法。例如，许多 Web 用户认为如果浏览器返回一个在最近几分钟内正确性未经验证的缓存文档是可以接受的。然而，在许多情况下必须确保高度的一致性，比如在进行电子股票交易时就是如此。高度一致性的关键问题是，对某个副本的更新必须立即传播到其他所有副本上。此外，如果两个更新操作并发执行，一般会要求每个副本都按照相同的次序进行更新。对这些情况的处理一般需要采取某些全局同步机制。但是，要以一种可扩展的方式来实现这样的机制是极为困难的，甚至是不可能的。因此，通过复制来提高可扩展性将会引入另一种本质上不可扩展的解决方案。我们将在第 6 章中详细讨论复制和一致性问题。

1.3 分布式系统的类型

在开始介绍分布式系统的原理之前，让我们首先来看看分布式系统的不同类型。下面我们介绍了分布式计算系统、分布式信息系统和分布式嵌入系统之间的差别。

1.3.1 分布式计算系统

分布式系统一个重要的分类就是用于高性能计算任务的系统。粗略地说，它可以分成两个子分组。在**集群计算**（cluster computing）中，底层硬件是由类似的工作站或 PC 集组成，通过高速的局域网紧密连接起来的。而且，每个结点运行的是相同的操作系统。

在**网格计算**（grid computing）中，情况就很不相同了。组成分布式系统的这种子分组通常构建成一个计算机系统联盟，其中的每个系统归属于不同的管理域，而且在硬件、软件和部署网络技术上也差别很大。

1.3.1.1 集群计算系统

在个人计算机和工作站的性价比提高后，集群计算系统变得很普遍了。在某些情况下，使用现有的技术，通过在高速网络上挂接相对简单的计算机集，就可以构建成一个超级计算机，这在经济上和技术上都是很有吸引力的。在所有虚拟情况下，集群计算系统都可用于并行程序设计，即单个程序可以在多台计算机上并行地运行。

集群计算系统一个著名的示例就是基于 Linux 的 Beowulf 集群，其常见的设置如图 1.6 所示。每个群都是由一个计算机结点集组成的，它们可以由单个主结点来控制和访问。主结点通常负责对特定并行程序的结点定位，维护已提交工作的队列，并为系统用户提供一个接口。因此，主结点实际上运行的是程序运行和集群管理所需的中间件，而计算机结点往往只需要一个标准的操作系统。

这种中间件的重要部分是由运行并行程序所需的库构成的。正如我们将在第 4 章中所介绍的那样，很多的这种库只是高效地提供高级的基于消息的通信工具，但不能负责故障处理、安全等问题。

在 MOSIX 系统中（Amar 等 2004）提出了一种对称方法，它是上面这种异构组织方式另一种形式。MOSIX 提供了集群的**单系统镜像**（single-system image），这意味着，集群计

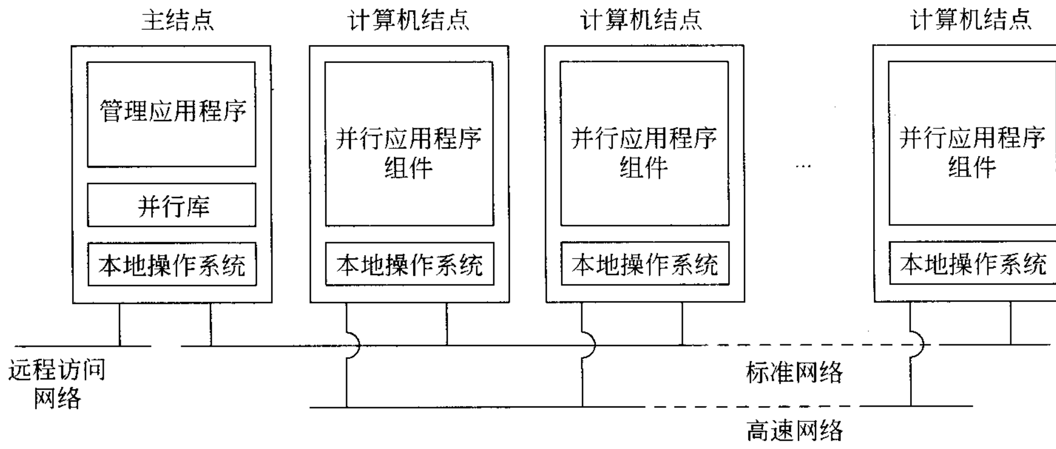


图 1.6 集群计算系统的一个示例

算机提供了最大限度的分布式透明性，对某个进程而言，它就像是单个的计算机。正如我们已经介绍的那样，在所有环境下都提供这种镜像是不可能的。在 MOSIX 中，透明度是通过允许进程在集群的结点之间动态和优先迁移来提供的。进程迁移允许用户在任意结点开始应用程序的运行，然后，(例如，为了有效利益资源)又可以透明地迁移到其他结点上去。第 3 章我们将回过头来继续讨论进程迁移问题。

1.3.1.2 网格计算系统

集群计算系统一个突出特征是它的同构性。在大多数情况下，集群中的计算机都是相同的，它们都具有相同的操作系统，都通过同一网络连接起来。而网格计算系统则具有高度的异构性：其硬件、操作系统、网络、管理域和安全策略等都不尽相同。

网格计算系统一个关键的问题是，如何把来自不同计算机组织的资源集中起来，使一组人或机构进行协调工作。这种协调是以**虚拟组织**(virtual organization)的方式来实现的。属于同一虚拟组织的人，具有访问提供给该组织的资源的权限。通常，这种资源是由计算机服务器(包括超级计算机，它可能是由集群计算机实现的)、存储工具和数据库组成的。另外，也可以提供特殊的网络设备，如望远镜、传感器等。

鉴于它的这种特性，用于实现网格计算的软件，从为不同的管理域提供资源访问，发展到为属于某个虚拟组织的用户或应用程序提供资源访问。由于这个原因，网格计算系统的关注重点往往是体系结构的问题。图 1.7 是(Foster 等 2001)提出的一种体系结构。

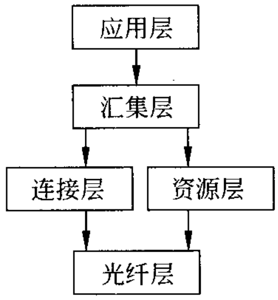


图 1.7 网格计算系统的分层体系结构

该体系结构由 4 层组成。最低的**光纤层**(fabric layer)在特定站点提供对局部资源的接口。注意，这些接口都进行了定制，以允许在某个虚拟组织中实现资源共享。通常，它们还提供查询资源状态和容量的功能，以及用于进行实际的资源管理(如加锁资源)的功能。

连接层(connectivity layer)由通信协议组成，用于支持网格事务处理，延伸多个资源的使用。例如，用于在资源之间传输数据或从远程地点访问资源的协议。另外，连接层还含有安全协议，用于进行用户和资源的认证。注意，在很多

情况下，使用者不需要认证，而是由程序代替用户进行认证。在这种情况下，用户给程序的委托权就是一个很重要的功能，需要在连接层中支持。在讨论分布式系统的安全性时，我们会扩展讨论这种委托权。

资源层(resource layer)负责管理单个资源。它使用由连接层提供的功能，直接调用对光纤层可用的接口。例如，该层可提供用于获取某个资源的配置信息，或者执行某个操作，如生成一个进程或读取数据等。因此，资源层可看作是负责访问控制，因而依赖于连接层的认证功能。

接下一层是**汇集层**(collective layer)。该层负责处理对多个资源的访问，通常由资源分派、把任务分配和调度到多个资源以及数据复制等服务组成。与连接层和资源层不同，这些层是由相对较小、较标准的协议集组成的，而汇集层可能由很多用于不同目的的不同协议组成，从它可以为虚拟组织提供很多的服务就可以反映出这一点。

最后，**应用层**(application layer)由应用程序组成，这些应用程序在虚拟组织中运行，它们可以充分利用网格计算环境。

通常，汇集层、连接层和资源层形成了称为网格中间件层的核心。这些层提供了对资源的访问和管理，而这些资源是分散在多个地点的。从中间件的角度来看，一个重要的因素是网格计算的站点(或管理单元)表示法很常见。这种流行趋势在**面向服务的体系结构**(service-oriented architecture)中逐步得到了强调。目前，这导致了出现另一种体系结构的定义的出现，这就是**开放式网格服务体系结构**(Open Grid Services Architecture, OGSA)。这种体系结构由不同的层和组件组成，使得它相当复杂。看来，复杂是标准化过程的必然结果。有关 OGSA 的详细讨论可以参见(Foster 等 2005)。

1.3.2 分布式信息系统

分布式系统另一个重要的分类是在组织中的应用，这些组织面临大量的网络应用，但这些应用之间的互操作性很差。很多现有的中间件解决方案是使用一种基础设施，它可以把这些应用集成到企业范围内的信息系统中(Bernstein 1996; Alonso 等 2004)。

我们可以按集成程度进行分类。在很多情况下，一个网络应用由一个服务器组成，它负责运行应用程序(通常包括一个数据库)并使得它对远程程序(称为**客户端**(client))可用。这种客户端可以发送一个请求给服务器，用于运行某个操作，然后会送回一个响应。最低级的集成就是允许客户端把多个请求(可能是发往多个服务器的)封装成单个较大的请求，并使这个较大的请求作为一个**分布式事务处理**(distributed transaction)来运行。其关键点是要么所有请求都被运行，要么所有都不被运行。

随着应用程序变得更加复杂，并逐步分割成各自独立的组件(最明显的是把数据库组件与事务处理组件分开)，显然，通过让应用程序直接与其他程序进行通信，也可以实现集成。这就是现在大型企业所关注的**企业应用集成**(enterprise application integration, EAI)。下面将主要介绍这种分布式系统的两种形式。

1.3.2.1 事务处理系统

为清楚起见，这里我们主要以数据库应用为例进行介绍。在实际使用中，对数据库的操

作实际上是以**事务处理**(transaction)的形式进行的。对使用事务处理的程序要求有特定的原函数,即必须能被底层的分布式系统或编程语言的运行时系统所支持。图 1.8 显示了事务处理原函数的典型示例。有哪些原函数取决于在事务处理中使用的是什么类型的对象(Gray 和 Reuter 1993)。在邮件系统中,原函数包括发送、接收和转发邮件。在财务系统中则可能完全不同。但是,通常都有 READ 和 WRITE。通常来说,在事务处理中还可以进行过程调用等。特别地,我们说,**远程过程调用**(remote procedure call, RPC,也就是说,这种过程可以调用远程服务器)通常也可以封装到一个事务处理之中,称为事务处理 RPC(transactional RPC)。我们将在第 4 章展开讨论 RPC。

原函数	说明
BEGIN_TRANSACTION	标识一个事务处理的开始
END_TRANSACTION	终止事务处理并试图提交
ABORT_TRANSACTION	杀死事务处理并恢复旧值
READ	从文件、表或其他地方读取数据
WRITE	往文件、表或其他地方写入数据

图 1.8 事务处理的原函数示例

BEGIN_TRANSACTION 和 END_TRANSACTION 用于限制事务处理的范围。位于这两者之间的操作形成了事务处理的主体。事务处理的突出特征是要么所有这些操作都被运行,要么都不运行。这些操作可以是系统调用、库过程或实现语言的分支语句等。

事务处理这种全是或全非(all-or-nothing)的属性是其 4 个突出属性之一。具体地说,事务处理具有以下 4 个特性。

(1) 原子性:对外部来说,事务处理是不可见的。

(2) 一致性:事务处理不会违反系统的不变性。

(3) 独立性:并发的事务处理不会相互干扰。

(4) 持久性:事务处理一旦提交,所发生的改变是永久性的。

这些属性通常用它们的首字母来表示:ACID。

所有事务处理展现的第一个主要属性就是它们具有**原子性**(atomic)。这个属性确保每个事务处理要么完全被运行,要么完全不被运行,而且,如果被运行,就是在单个不可分割的、瞬间的动作中进行的。当某个事务处理处在进程中,其他进程(不管它们本身是否包括在事务处理中)都不能看到任何的中间状态。

第二个属性就是**一致性**(consistent)。这意味着,如果系统具有某些必须保持的不变性,而且在事务处理之前就保持着,那么,在事务处理之后也将继续保持着。例如,在银行系统中,一个关键的不变性就是金额的守恒定律。在每次内部转账之后,银行的金额数量必须与转账之前的相等,但在事务处理的某个瞬间,这种不变性可能会被破坏。但是,这在事务处理的外部是看不见的。

第三个属性就是事务处理具有**独立性**(isolated)或**可序列化性**(serializable)。这意味着,如果有两个或多个事务处理同时运行,对它们相互之间或对其他进程来说,最后的结果看上去就像所有这些事务处理是按某种顺序依次运行的。

第四个属性就是事务处理具有**持久性**(durable)。它表示,事务处理一旦提交,不论发生了什么,该事务处理都将往前进行下去,而且其结果是永久性的。事务处理提交后,不能

回滚结果或丢弃结果(第8章将展开讨论持久性)。

至此,事务处理是在单个的数据库上进行定义的。**嵌套式事务处理**(nested transaction)是由多个子事务处理构成的,如图1.9所示。最顶层的事务处理分支成其下一级的事务处理(这些子事务处理与其他事务处理在不同的机器上并行地运行),以获得较高的性能或简化程序设计。而这些下一级的事务处理又可以运行一个或多个子事务处理,或者再分支成其自己的下一级事务处理。

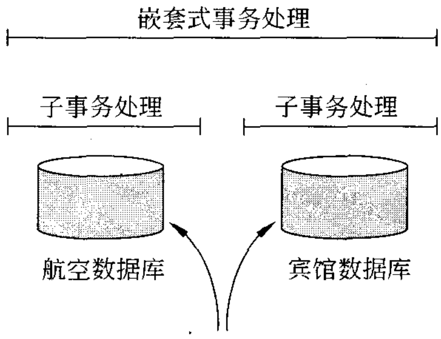


图1.9 嵌套式事务处理

子事务处理带来了一个微妙但很重要的问题。假设一个事务处理并行地启动了多个子事务处理,并提交了其中的一个,使其结果对父事务处理是可见的。在后面的计算中,父事务处理发生中止,并把系统恢复到最顶层事务处理启动之前的状态。因此,所提交的子事务处理的结果必须仍然能回滚。这样,上面所说的持久性只适用于最顶层的事务处理。

由于事务处理可以嵌套任意多层,就需要有深思熟虑的管理方法来保证正确性。这在语义上是很清楚的。当启动任何事务处理或子事务处理时,从概念上来说,就给了它一份整个系统的所有数据的副本,它可以按自己的需要进行操作。如果它中止了,只是丢弃它自己的那份副本,就像它从没有存在过一样。如果它提交了,那么它的数据将取代其父事务处理的数据。因此,如果某个子事务处理提交了,然后又启动了一个新的事务处理,第二个事务处理就可以看见第一个事务处理所产生的结果。同样,如果更高层的事务处理中止了,那么它的所有子事务处理也必须中止。

在分布式系统中,嵌套式事务处理很重要,因为它们提供了一种很自然的方法,可以把一个事务处理分布到多台机器上。它们是按照原始事务处理工作的逻辑分支进行的。例如,对于一个规划旅行的事务处理,它有三个不同的航班需要预订,于是可以把它按逻辑分割成三个子事务处理。每个子事务处理可以分别进行管理,且与其他两个是相互独立的。

在企业中间件系统的早期,负责处理分布式(或嵌套式)事务处理的组件在服务器或数据库级上构成了集成应用的核心。这种组件称为**事务处理监视器**(transaction processing monitor),或简称为**TP监视器**(TP monitor)。其主要任务是通过提供如图1.10所示的事务处理编程模式,允许应用程序访问多个服务器/数据库。

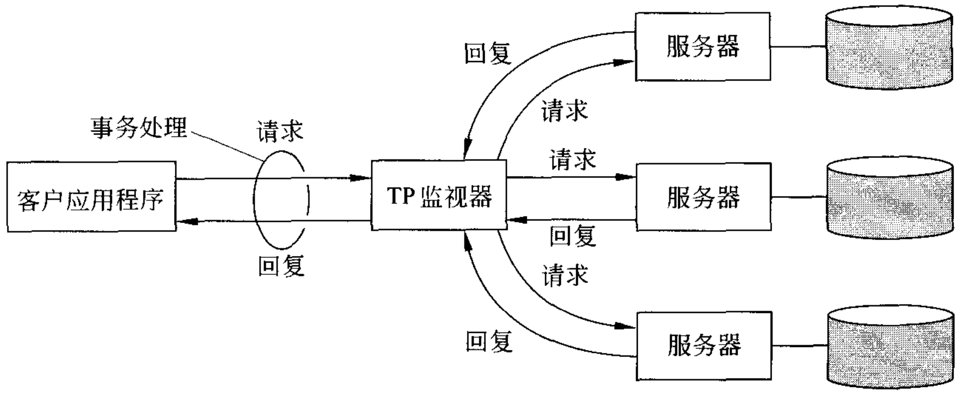


图1.10 分布式系统中TP监视器的作用

1.3.2.2 企业应用集成

如前所述，应用程序与其关联的数据库越独立，就越需要有工具来把这些与数据库无关的应用程序集成到一起。特别是，应用程序组件之间应可以直接进行通信，而不仅限于通过由事务处理系统所支持的请求/回复。

应用程序间的通信需求导致出现了不同的通信模式，我们将在本书中分别详细介绍。最主要的思想是已有的应用程序间应可以直接交换信息，如图 1.11 所示。

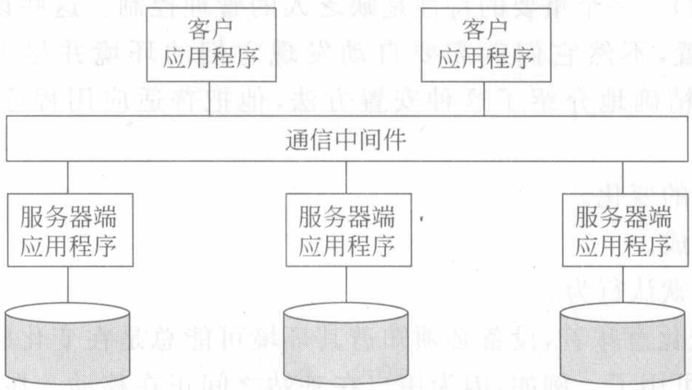


图 1.11 在企业应用集成中，中间件作为一种通信工具

现在已经有了几种通信中间件了。**远程过程调用**(remote procedure call, RPC)就是这样一种应用程序组件，它可以通过一个本地过程调用，有效地往另一个应用程序组件发送一个请求。本地过程调用可以使其请求封装成一个消息，并发送给被调用者。同样，过程调用的结果将被发送回来并返给应用程序。

随着对象技术越来越流行，已经开发出了调用远程对象的技术，这就是**远程方法调用**(remote method invocation, RMI)。RMI 与 RPC 实质上是相同的，只不过它操作的是对象而不是应用程序。

RPC 和 RMI 都有一个不足之处，那就是，在通信的时候，调用者和被调用者都必须正在运行之中。而且，它们必须正确地知道如何引用对方。这种严格配对的特点往往会形成一个严重的缺点，这导致了**面向消息的中间件**(message-oriented middleware, MOM)的产生。在这种情况下，应用程序只需发送一个消息给逻辑意义上的联系点(通常用目标来描述)。同样，应用程序也可以表明它对某些类型的消息感兴趣，通信中间件就会把这些消息移交给这些应用程序。这种所谓的**发布/订阅**(publish/subscribe)系统就构成了一种重要且可扩展的分布式系统。我们将在第 13 章中详细介绍。

1.3.3 分布式普适系统

到目前为止，我们所介绍的分布式系统最大的特点就是它们的稳定性：结点是固定的，具有或多或少的持续性和高质量的网络连接。通过本书将介绍的一些技术，这种稳定性有了一定的扩展，目标是获得分布式透明性。例如，故障隐藏与恢复技术给人的印象是，只有极少的时候会发生错误。同样，我们可以隐藏起某个结点的真实网络位置，使得用户和应用

程序以为结点是原地不动的。

但是，随着移动和嵌入式计算设备的引入，事情就变得很不一样了。我们现在面临的分布式系统是，其不稳定性是其默认的行为。在这种分布式普适系统中的设备，往往具有体积小、由电池供电、可移动以及只有一个无线连接等特征（但并不是所有设备都同时具有这些特征）。而且，这些特性并不需要严格地解释，正如现代智能电话的可行性中所说明的那样（Roussos 等 2005）。

正如其名字所表示的那样，分布式普适系统是我们周围环境的一部分（因此，它往往与生俱来就是分布式的）。一个重要的特性是缺乏人的管理控制。这些设备充其量可以由它们的所有者进行配置，不然它们就需要自动发现它们的环境并尽可能“安置”于其中。（Grimm 等 2004）更精确地介绍了这种安置方法，他把普适应用程序的需求归纳为以下三种：

（1）接受上下文的变化。

（2）促使自主合成。

（3）认可共享为默认行为。

接受上下文的变化意味着，设备必须知道其环境可能总是在变化的。一种最简单的变化是，发现网络不再可用了。例如，因为用户在基站之间正在移动。在这种情况下，应用程序应有所反应，可能要自动地连接到另一个网络中，或者采取其他恰当的动作。

促使自主合成指的是，普适系统中的很多设备可能被不同的用户以不同的方式使用。因此，通过用户或自动化的（但可控的）干预，它应该可以很容易地配置运行在设备上的应用程序簇。

普适系统一个非常重要的方面就是，设备经常要加入到系统中，以便访问（或提供）信息。这就要求有容易读取、存储、管理和共享信息的方法。由于设备的不连续性和不断改变的连接性，可访问信息的存储空间也非常有可能总是变化的。

（Mascolo 等 2004）与（Niemela 和 Latvakoski 2004）得到了类似的结论：在可移动性中，设备支持对本地环境的易适应性和与应用程序相关的易适应性。它们必须能有效地发现设备并相应地做出动作。从这些需求可以清楚地看出，在普适系统中，并不存在真正的分布式透明性。事实上，数据、过程和控制的分布性为这些系统所固有的，由于这个原因，与其试图隐藏它，还不如就暴露出来。下面让我们来看看普适系统的一些具体例子。

1.3.3.1 家庭系统

普适系统的一种日益流行的（可能也是受约束最少的）类型是家庭网络系统。这些系统通常由一台或多台个人计算机组成，但最重要的是，它把常见的电子设备，如电视机、音频和视频设备、游戏设备、（智能）电话、PDA 和其他个人可穿戴设备集成为单个系统。另外，我们希望把所有设备，如厨房设备、监视摄像、时钟和灯光控制器等也集成到单个分布式系统中。

从系统的角度来说，在普适系统变成现实之前，有几个需要解决的挑战问题。一个重要的问题是，这种系统必须是完全自我配置和自我管理的。如果这种系统的组件容易出错（现今使用的很多设备就容易出错），就不可能期望终端用户会愿意或者有能力使用分布式家庭系统。通过**通用即插即用**（Universal Plug and Play，UPnP）标准，很多问题已经得到解决。

利用通用即插即用标准，设备可以自动获得 IP 地址，从而可以相互发现设备(UPnP Forum 2003)。但是，光有这些还不够。例如，如果没有手工干预，就不清楚设备中软件和固件怎样能容易地进行更新，或者什么时候更新，怎样才能不与其他设备的兼容性发生冲突。

另一个紧迫的问题是对“个人空间”的管理。一个家庭系统是由许多可共享的个人设备组成的，家庭系统中的数据也要受到共享限制，因此需要注意如何实现这种个人空间。例如，Alice 的个人空间可能由她的日程表、家庭相册、日记、她购买的音乐和视频组成。这些个人资产按这样一种方式进行存储：Alice 可以随时访问它们。而且，这种个人控制应该可以(或临时)被其他人访问，例如，当她需要制定一个商业约定时。

幸运的是，事情可能会变得简单些。很久以来，人们总是认为，家庭系统的个人空间是可以跨越不同设备的分布式的。显然，这种发散性易于产生同步性问题。但是这些问题可能由于硬盘空间的快速增大及硬盘尺寸的变小而有所缓解。配置一个几 T 存储空间的个人计算机已不是一个问题了。与此同时，可移动的硬盘也具有好几百 G 的存储空间了，而且可以放置在相对较小的可移动媒体播放器中了。随着这些能力的不断增长，我们或许可以看到，普适家庭系统可能采用这样一种体系结构，用一个机器作为主设备，其他的固定设备只是为用户提供一个方便的接口。这样，个人设备可以存储每日所需的信息，永远也不用担心存储空间不足。

但是，有了足够的存储空间并不意味着就解决了个人空间管理的问题。可以存储大量数据加重了相关数据的存储以及日后查找它的问题。我们将不断地看到，普适系统，如家庭网络，装备了 recommender 软件，这种程序可以参考其他用户所存储的内容，以便标识出它们的相似性，然后得出哪些内容存放在个人空间中。一个令人感兴趣的因素是，recommender 程序所需的信息量往往很小，足以让它们在 PDA 上运行(Miller 等 2004)。

1.3.3.2 电子健保系统

普适系统另一个重要且即将出现的类型是(个人)**电子健保系统**(electronic health care)。随着医疗费用的增长，正在开发新的设备，用于监视个人的安康，并在需要的时候与医生联系。这些系统的一个主要目标就是防止人们住院治疗。

个人健保系统往往配有各种传感器，组成一个(无线)**身体局域网**(body-area network，BAN)。这种网络一个重要的问题是，在最坏的情况下也只会妨碍一个人。当人移动时，不用绳子(即电线)与固定设备相连，这种网络应仍可以操作。

这种需求导致了两种组织形式的出现，如图 1.12 所示。在第一种形式中，中心集线器为身体局域网的一部分，需要时用于收集数据。这种方案的优点是，该集线器也可以管理身体局域网。在第二种形式下，通过无线连接，身体局域网总是挂在一个外部网络之中，并通过无线连接来发送监视数据。在这种情况下，需要用单独的技术来管理身体局域网。当然，同样还需要有与医生或其他人之间的连接。

从分布式系统的角度来看，我们很快就会面临一些问题，如：

(1) 监视数据应保存在哪里，如何保存？

(2) 如何防止关键数据的丢失？

(3) 生成和广播警告信息需要什么基础设施？

(4) 医生如何提供在线反馈？

（5）如何实现监视系统的健壮性？

（6）会有些什么安全问题？如何采取正确的策略？

与家庭系统不同，我们不能期望普适健保系统的体系结构是在单个服务器系统中移动，由一个监视设备负责基本的功能。相反，出于效率的原因，要求设备和身体局域网能支持**网络内数据处理**(in-network data processing)，这意味着，比如，监视数据必须在永久存储或发送给医生之前就收集起来。与分布式信息系统不同，对这些问题还没有清楚的答案。

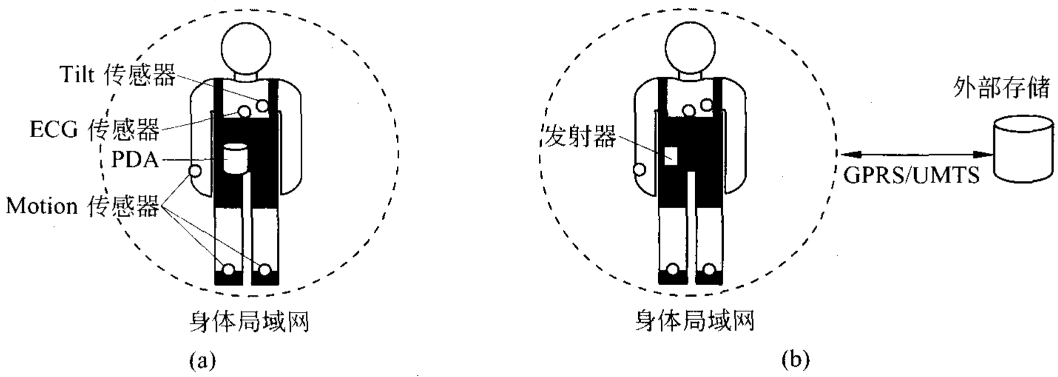


图 1.12 电子保健系统

(a) 本地集线器；(b) 无线连接

1.3.3.3 传感器网络

我们要介绍的最后一个普适系统示例是**传感器网络**(sensor network)。在很多情况下，这些网络形成了普适技术的一部分，而且我们将看到，传感器网络的很多解决方案都归结于普适应用。从分布式系统的角度来说，使得传感器网络让人感兴趣的是它们用于处理信息的所有情况。从这种意义上来说，它们不只是提供通信服务，而这正是传统计算机网络所做的一切。(Akyildiz 等 2002)从网络技术的角度给出了传感器网络的一个概述。(Zhao 和 Guibas 2004)则给出了更趋于面向系统的传感器网络介绍。与传感器网络紧密相关的是**网状网络**(mesh network)，这种网络形成了一个固定的结点集，通过无线链接进行通信。这些网络是很多媒体可扩展的分布式系统的基础。(Akyildiz 等 2005)给出了网状网络的一个概述。

传感器网络通常由成千上万个相对较小的结点组成，每个结点配有一个感应设备。很多传感器网络使用无线通信，这些结点经常是由电池来供电。它们这种有限的资源、有限的通信能力，以及有限的电力消耗要求，使得在设计标准中，效率是优先考虑的。

把传感器网络看作是分布式数据库，就可以清楚地明白它与分布式系统之间的关系。当意识到很多传感器网络的部署是用于测量和监视应用的(Bonnet 等 2002)，这种观点是很普遍而且很容易理解的。在这些情况下，一个操作者只需发出一个诸如“在一号高速路上往北方向的车流量如何?”的查询，就可以从网络中抽取出信息。这种查询类似于传统数据库的查询。在这种情况下，其答案将需要位于高速路旁的很多个传感器的相互协作来提供，而其他的传感器则不用触及到。

要把一个传感器网络组织成一个分布式数据库，有两种极端的情况，如图 1.13 所示。第一种是传感器不会相互协作，而是往位于操作者地点的集中式数据库发送它们的数据。

另一种情况是把查询转发给相关的传感器,让每个传感器计算出一个答案,要求操作者明智地聚集所返回的答案。

这两种方法都不吸引人。第一种要求传感器把所有的测量数据通过网络发送,这可能会浪费网络资源和通信能力。第二种方法也可能会有浪费,因为它弃用了传感器的聚集能力(这种能力可以使得只需更少的数据返回给操作者)。这里所需要的是用于网络内数据处理工具,正如我们在普适健保系统中所遇到的那样。

网络内数据处理可以以多种方法来完成。一种显而易见的方式是沿着包围了所有结点的树,把查询转发给所有传感器结点,然后把所生成的结构聚集起来返回给发起者所在的根结点。在树的两个或多个分支交汇处,就会发生聚集操作。这种方案听起来很简单,但它会产生以下一些难题:

(1) 如何在传感器网络中(动态地)创建一个有效的树?

(2) 聚集结果是如何产生的? 它是可控吗?

(3) 当网络链接出现故障时,会发生什么?

这些问题在 TinyDB 中得到部分解决。TinyDB 为无线传感器网络实现了一个声明式的(数据库)接口。从本质上说,TinyDB 可以使用任何基于树的路由算法。中间结点可以收集和聚集其子分支的结果,并连同它自己的结果一起,发送给根结点。为了使得事情更有效,查询需要持续一段时间,以便仔细地调度操作,从而使得传感器网络的资源和电力得到最佳使用。详细信息请参见(Madden 等 2005)。

但是,当查询可以从网络中的不同地点发起时,像 TinyDB 那样使用单个根结点的树并不是很有效。一种方法是,传感器网络可能配置一些特殊的结点,计算结果及与这些结果相关的查询都可以转发给这些特殊结点。一个简单的示例是,有关温度查询及结果的收集地点与湿度测量的不同。这种方法与发布/订阅系统的表示方法是直接对应的。我们将在第 13 章展开介绍。

本章小结

分布式系统是由自主计算机组成的,这些计算机一起工作,给人的外观是一个单一连贯的系统。一个重要的优点是,它们可以很容易地把运行在不同计算机上的不同应用程序集成到单个系统中。另一个优点是,如果设计恰当,分布式系统可以随底层网络的大小而扩展。这些优点往往带来的代价是更复杂的软件、性能的降低,而且往往削弱了系统的安全性。但是,创建和安装分布式系统还是有意义的。

分布式系统的目标往往是要隐藏掉有关分布式处理、数据和控制的复杂性。但是,这种分布透明性不仅导致较差的性价比,而且在实际情况中,是不可能完全实现的。事实是,需要在实现分布式透明性与使系统复杂化之间进行平衡。

事情远非这么简单,很多开发人员最初假设底层网络根本上是错误的。然后,当这些假设行不通时,可能会发现要隐藏掉那些不想要的行为是很困难的。一个常见的示例是,假设网络延时不是重要因素。后来,当要把一个已有系统移植到广域网时,隐藏起来的系统延时可能会严重影响系统的初始设计。其他陷阱还有假设网络是可靠的、静态的、安全的以及同构的。

已有的分布式系统根据所支持的计算方法、信息处理和普适性,可分为不同的类型。分

布式计算系统通常用于高性能应用。这种应用起源于并行计算领域。分布式系统的一个大类可以在传统的办公环境找到。在这种环境中，数据库起着很重要的作用。通常，事务处理系统部署在这些环境中。最后是一种正在涌现的分布式系统类型，其中的组件比较小，系统以一种自主的方式组成，而且大多数不再是通过系统管理员来管理。最后一种类型的分布式系统通常表现为无所不在的计算环境。

习题

1. 分布式系统的另一个定义，它是各自独立的计算机的集合，这些计算机看起来像是一个单个的系统，也就是说，它对用户是完全隐藏的，即使它有多个计算机也是如此。请给出一个示例。
2. 中间件在分布式系统中扮演着什么角色？
3. 很多网络系统组织成后端办公系统和前端办公系统。这种组织方式是如何满足分布式系统要求的？
4. 请解释（分布式）透明性的含义，并且给出各种类型的透明性示例。
5. 在分布式系统中，为什么有时难以隐藏故障的发生以及故障恢复过程？
6. 为什么有时候最大程度地实现透明性并不总是好的？
7. 什么是开放的分布式系统？开放性带来哪些好处？
8. 请对可扩展系统的含义做出准确描述。
9. 可以通过应用多种技术来取得可扩展性。都有哪些技术。
10. 请解释一下什么是虚拟组织，并请给出如何实现这种组织的一些提示。
11. 当一个事务处理被异常中断时，将会恢复到它以前的状态，就像这个事务处理从没有发生过一样。我们这么说其实是不对的。请给出一个无法恢复以前状态的示例。
12. 运行嵌套式事务处理需要某种形式的协作。请解释一下协作者应真正做些什么事情？
13. 我们讨论过，对普适系统来说，分布式透明性可能并不存在。这句话并不是对所有的透明性都是正确的。请给出一个示例。
14. 我们已经给出了一些分布式普适系统的示例：家庭系统、电子健保系统和传感器网络。请再给出一些这样的示例。
15. **（实验）**请草拟一个家庭系统，它有一个单独的媒体服务器，允许无线客户访问。无线客户可以连接到（模拟）音频/视频设备，并把数字化的媒体流转换为模拟输出。服务器运行在单独的机器上，可能连接到因特网上去，但没有连接键盘或监视器。

第2章

体系结构

分布式系统往往是由各种复杂的系统组成，其组件按定义分散在多台机器之中。要掌握这些复杂性，关键是恰当地组织好这些系统。察看分布式系统的组织结构的方法有多种，其中常见的一种是区分软件组件集的逻辑组织和实际物理实现的差别。

分布式系统的组织结构主要是指组成该系统的软件组件。这种**软件体系结构**(software architecture)告诉我们不同的软件组件是如何组织的，它们应如何相互作用。本章我们首先来看看一些常见的组织(分布式)计算机系统的实用方法。

分布式系统的真正实现要求我们真实地把软件组件放置在真实的机器上。这样做的选择有多种。软件体系结构的最终实例又称为**系统体系结构**(system architecture)。本章我们将看看传统的集中式体系结构，其中，单个服务器实现了大多数的软件组件(及其功能)，而远程客户使用简单的通信方法就可以访问该服务器。另外，我们还将察看非集中式体系结构，其中的每个机器起着同等的作用，就像混合组织结构一样。

正如在第1章中所介绍的那样，分布式系统一个重要的目标是，通过提供一个中间件层，把应用程序与底层平台分开。采用这样一层是一个重要的体系结构决策，其目的是提供分布式透明性。然而，要获得透明性是需要付出代价的，需要不同的技术来使得中间件为可适应的。本章将介绍一些更为常见的实用技术，它们本身也影响着中间件的组织结构。

通过使分布式系统监视自己的行为，当需要时采取适当的措施，就可以获得适应性。这就导致了**自治系统**(Autonomic Systems)的出现。这种分布式系统经常组成反馈控制循环的形式，形成了系统设计中一个重要的体系结构元素。本章将用一节的篇幅来介绍自治分布式系统。

2.1 体系结构的样式

我们首先从把分布式系统的逻辑组织结构看成为软件组件(又称为软件体系结构)(Bass 等 2003)来开始其体系结构的讨论。有关软件体系结构的研究已经相当成熟了，现在被人们普遍接受的是，大型系统能否成功开发的关键是体系结构的设计和采用。

由于这里的讨论需要，**体系结构样式**(Architecture Style)的表示法很重要。这种样式是根据组件、组件之间相互的连接方式、组件之间的数据交换以及这些元素如何集成到一个系统中来定义的。**组件**(Component)是一个模块单元，它具有并且可以提供良好定义的接

口,在其环境中是可替换的(OMG 2004b)。正如我们下面要讨论的,分布性系统中有关组件的一个重要问题是可替换性,以及能否提供我们所需的接口。一个稍为更难掌握的概念是**链接器**(Connector),通常把它描述成这样一种机制:在组件之间传递通信、使组件相互协调和协作(Mehta 等 2000;Shaw 和 Clements 1997)。例如,一个链接器可以利用(远程)过程调用、消息传递或数据流来形成。

根据组件和链接器的使用,就可以出现不同配置,从而也就可以划分成不同的体系结构样式。现在已认可的样式有多种,分布式系统中最重要的有:

(1) 分层体系结构;

(2) 基于对象的体系结构;

(3) 以数据为中心的体系结构;

(4) 基于事件的体系结构。

分层体系结构的思想很简单:组件组成了不同的层,其中 L_i 层中的组件可以调用下面的层 L_{i-1},如图 2.1(a)所示。此模型被网络通信广泛采用:我们将在第 4 章中简要介绍。一个关键因素是,其控制系统是从一层到另一层的:请求是从上往下,而请求结果则从下往上。

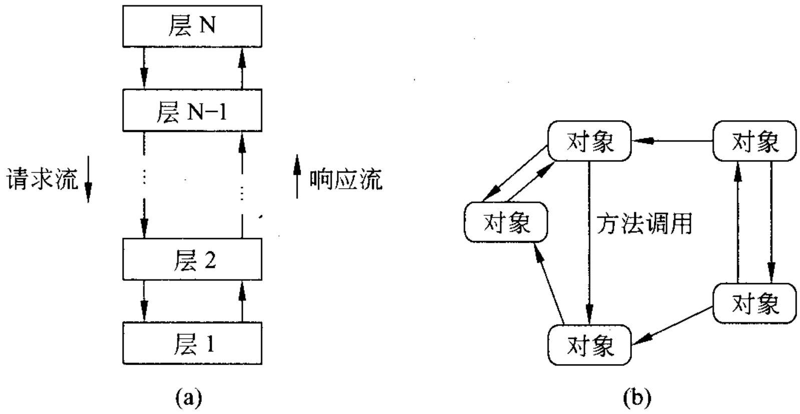


图 2.1 分层(a)和基于对象的(b)体系结构样式

基于对象的体系结构(object-oriented architecture)是一种很松散的组织结构,如图 2.1(b)所示。基本上,每个对象都对应一个组件,这些组件是通过(远程)过程调用机制来连接的。毫不奇怪,这种软件体系结构与我们上面介绍的客户-服务器系统体系结构很匹配。分层和基于对象的体系结构仍然是大型软件系统最重要的样式(Bass 等 2003)。

以数据为中心的体系结构(data-centered architecture)是从这种思想中发展而来的:进程通信需要通过一个公用(被动或主动的)仓库。对分布式系统,认为这种体系结构与分层和基于对象的体系结构同等重要存在争论。例如,已经开发了很多网络应用程序,它们依赖一个共享的分布式文件系统,其中的所有通信都是通过文件虚拟发生的。同样,基于 Web 的分布式系统(我们将在第 12 章中展开讨论)大部分也是以数据为中心的:进程通信使用了共享的基于 Web 的数据服务。

在**基于事件的体系结构**(event-based architecture)中,进程基本上是通过事件的传播来通信的,事件传播还可以有选择地携带数据,如图 2.2(a)所示。对分布式系统来说,事件传播通常与**发布/订阅系统**(publish/subscribe systems)(Eugster 等 2003)有关。其基本思想

是，进程发布事件，然后，中间件将确保那些订阅了这些事件的进程才接收它们。基于事件的系统的主要优点是，进程是松散耦合的。从原理上来说，它们无需明确地引用对方。这又称为是**引用去耦**(referentially decoupled)的。

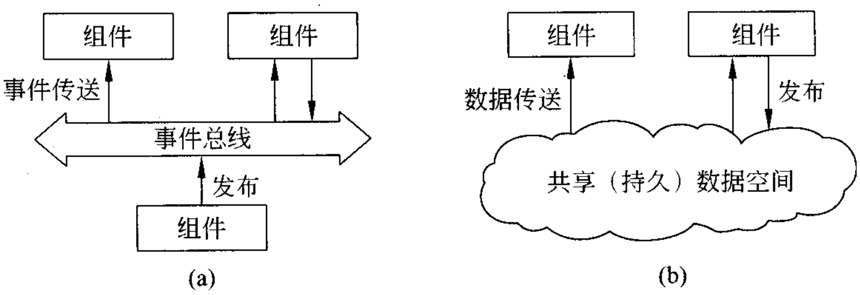


图 2.2　基于事件(a)和共享数据空间(b)的体系结构样式

基于事件的体系结构可以与以数据为中心的体系结构组合，形成**共享数据空间**(shared data spaces)。共享数据空间的基本是，进程也可以是去耦的：它们不需要两者在通信发生时都是活动的。而且，很多共享数据空间使用类似于 SQL 的接口与共享仓库交互，在这种情况下，使用描述符而不是明确引用就可以访问数据，就像是使用文件一样。第 13 章我们将深入介绍这种体系结构样式。

对分布式系统来说，软件体系结构之所以重要，是因为它们都要获得(合适程度的)分布式透明性。然而，正如我们已经讨论过的那样，分布式透明性要求与性能、容错性、程序设计难易性等进行平衡的。由于没有哪一种解决方法能够满足所有分布式应用程序的需求，研究人员已经放弃了这样一种思想：可以用单个分布式系统来满足所有 90%的可能情况。

2.2　系统体系结构

上面我们简要介绍了一些常见的体系结构样式，下面通过察看软件组件的放置位置，来看看有多少种分布式系统。确定软件组件、这些组件的交互以及它们的位置就是软件体系结构的一个实例，又称为**系统体系结构**(system architecture)(Bass 等 2003)。本节将介绍集中式和非集中式组织结构，以及各种混合形式。

2.2.1　集中式体系结构

人们在关于分布式系统的很多问题上很少意见一致，但在下面问题上很多研究人员和实践人员都认为：根据从服务器请求服务的客户来考虑问题，有助于我们理解和管理分布式系统的复杂性，这是一个好事情。

在基本的客户-服务器模型中，分布式系统中的进程分成了两组(可能会有重叠)。**服务器**(server)是实现特定服务的进程，例如，文件系统服务或数据库服务。客户(client)是通过往服务器发送请求来请求服务、然后等待服务器回复的进程。这种客户-服务器交互又称为**请求-回复行为**(request-reply behavior)，如图 2.3 所示。

如果底层网络很可靠(像很多局域网就是这样)，通过简单的无连接协议就可以实现客

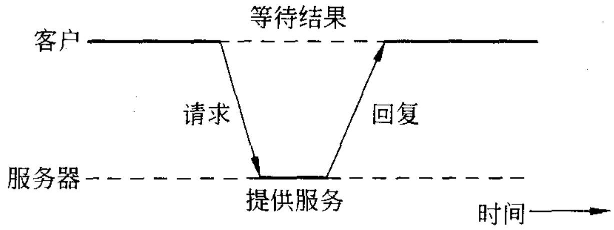


图 2.3 客户与服务器之间的常见交互

户和服务器之间的通信。在这些情况下，当客户请求一个服务时，它只需把要给服务器的消息封装，标明所要的服务以及所需的输入数据，然后就可以把消息发送给服务器。反过来，服务器总是等待进入请求，然后处理请求，并把结果封装到回复消息中，最后把该消息发送给客户。

使用无连接协议的一个明显优点是高效。只要消息不丢失或损坏，请求/回复协议就能很好地工作。但是，要使该协议能抵抗突发的传输故障是很困难的。我们唯一能做的大概是，当没有回复消息到来时，让客户程序重新发送请求。但问题是，客户无法检测原始请求消息是否丢失，也无法检测回复消息的传输是否发生了故障。如果回复消息丢失，那么，重新发送请求消息会导致执行两次相同的操作。如果某个操作可以重复多次而无害处，那么就称它为**幂等的**(idempotent)。由于有些请求是幂等的，而有些则不是，因此不能用某一个解决方法来处理消息的丢失问题。在第 8 章中我们将详细讨论传输故障的处理问题。

另外，很多客户-服务器使用可靠的基于连接的协议。尽管这种解决方法由于其相对较低的性能，不是完全适用于局域网，但在广域网中则可以很好地工作，因为在广域网中的通信是不可靠的。例如，所有因特网应用协议都是基于可靠的 TCP/IP 协议。在这种情况下，一旦客户要请求一个服务，它首先得在发送该请求之前创建一个到该服务器的连接。通常，服务器使用同一连接来发送回复消息，然后断开该连接。这里的困难是，创建和断开连接的开销较高，当请求和回复消息较小时更是如此。

2.2.1.1 应用分层

客户-服务器模型备受争议多年了。其主要问题之一是，如何在客户和服务器之间划分一个明显的界限。毫不奇怪，这里往往没有明显的界限。例如，分布式数据库的服务器可能会不断地作为一个客户来使用，因为它需要把请求转发给不同的文件服务器，以负责数据库表的实现。在这种情况下，数据库服务器基本上就只是完成进程查询了。

但是，假如很多客户-服务器应用程序的作用是支持用户对数据库的访问，那么，根据我们前面讨论的分层体系结构样式，就可以划分成如下三层：

(1) 用户接口层；

(2) 处理层；

(3) 数据层。

用户接口层含有直接与用户交互所需的一切，例如，显示管理。处理层通常包含有应用程序。数据层管理要使用的实际数据。

客户通常实现用户接口层。该层由允许终端用户与应用程序交互的程序组成。复杂的用户接口程序之间区别很大。

最简单的用户接口程序只有一个基于字符的屏幕。这种接口经常用在大型计算机环境中。在这些情况下，大型计算机控制着所有的交互，包括键盘和监视器，人们很难说这是一个客户-服务器环境。但是，在很多情况下，用户的终端能完成一个本地处理，如回显已输入的击键，或支持类似表单的接口，在其中，可以把整个项都编辑完成后，再发送给主计算机。

现在，即使是在大型计算机环境中，我们也可以看到更为先进的用户接口了。通常，客户计算机至少提供了一个图形显示，可以在其中使用弹出和下拉菜单，很多的屏幕控制就可以通过鼠标而不是键盘来完成了。这种接口的典型示例是使用在很多 UNIX 环境中的 X-Windows，以及早期为 MS-DOS PC 和 Apple Macintosh 开发的接口。

现代的用户接口提供了相当多的功能，允许应用程序共享单个图形窗口，以及使用窗口交换数据等。例如，要删除一个文件，通常可以把表示文件的图标移到表示垃圾桶的图标。同样，很多文字处理软件允许用户用鼠标把文档中的文本移到另一个地方去。第 3 章将介绍用户接口。

粗略地说，很多客户-服务器应用程序可以从三个部分构建而来：一部分负责处理与用户的交互，一部分负责操作数据库或文件系统，中间部分通常含有应用程序的核心功能。这个中间部分在逻辑上是位于处理层的。与用户接口和数据库不同，处理层没有太多的共同点。因此，我们将给出几个示例来介绍这一层。

作为第一个示例，让我们来看看因特网搜索引擎。如果忽略掉所有的动画标语、图像和其他窗口修饰，搜索引擎的用户接口非常简单：用户输入关键字字符串，然后显示出 Web 网页的列表。后端是由一个巨大的 Web 网页数据库组成，这些 Web 网页是预取的，且已加索引。搜索引擎的核心是把用户的关键字字符串转换成一个或多个数据库查询的程序。它可以把结果排名成一个列表，并把该列表转换成 HTML 页面列表。在客户-服务器模型中，这种信息检索部分往往放在处理层。如 2.4 显示了这种组织结构。

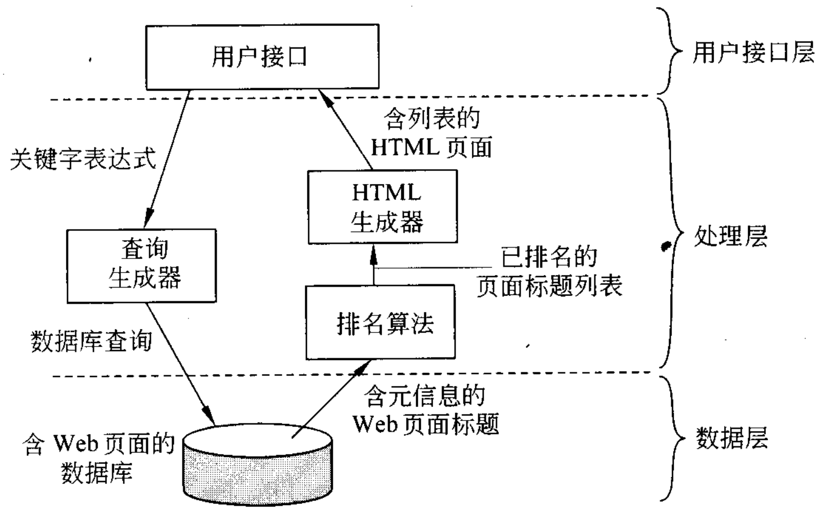


图 2.4　因特网搜索引擎简化成三个不同的层

第二个示例来看看金融决策支持系统。与搜索引擎类似，这种系统可以分成实现用户接口的前端，用于在财务数据中访问数据库的后端，以及位于这两者之间的分析程序。财务数据的分析需要很多统计和人工智能方面的复杂方法和技术。在某些情况下，金融决策支持系统甚至需要运行在高性能计算机上，以获得用户所期望的吞吐量和响应率。

最后一个示例来看一个桌面软件包，由字处理软件、表格应用程序、通信工具等组成。这种“办公”组件通常集成在一个通用的用户接口中，支持复合文档，可以操作用户本地目录中的文件（在办公环境中，本地目录往往放在远程文件服务器中）。在这个示例中，处理层由一个较大的程序集组成，每个程序具有相对简单的处理能力。

客户-服务器模型中的数据层含有用于维护实际数据的程序，应用程序可以在这些数据上进行操作。该层的一个重要属性是，其数据往往是**持久的**(persistent)，也就是说，即使应用程序没有运行，数据也会保存在某个地方以供日后使用。最简单的形式是，数据层由一个文件系统组成，但更为常见的是功能齐备的数据库。在客户-服务器模型中，数据层往往在服务器端实现。

除了存储数据外，数据层还负责保持不同应用程序之间的数据一致性。当数据库正在被使用时，维护数据的一致性意味着诸如表描述符之类的元数据、项约束和特定应用程序元数据也存储在这一层中。例如，在银行系统中，当用户的信用卡负债额度达到某一值时，就应生成一个通知。这种类型的信息可以通过数据库触发器来维护，在适当的时候激活触发器的处理程序。

在大多数基于商业的环境中，数据层组织成一个关系数据库。这里，数据的独立性很关键。数据组装成是与应用程序无关的，其方法是组织结构的改变不会影响应用程序，应用程序也不会影响数据组织结构。在客户-服务器模型中，使用关系数据库有助于把处理层与数据层分离开来，因为处理与数据是无关的。

然而，关系数据库并不总是理想的选择。很多应用程序的一个特征是，它们操作的是复杂的数据类型，这些数据类型用对象比用关系建模更容易。这种数据类型的示例包括从简单的多边形和圆到飞行器设计的表示，计算机辅助设计(CAD)系统也属于这种情况。

在这些情况下，数据操作以对象操作的方式表示就很容易，它使得通过基于对象或对象-关系数据库来实现数据层变得有意义。对象-关系数据类型已获得广泛认可，因为这些数据库是建立在广泛分散的关系数据模型上，同时又可以提供面向对象所具有的优点。

2.2.1.2 多层体系结构

到目前为止所讨论的三个逻辑层之间的差别，表明了物理上把客户-服务器应用分布到多个机器上的可能性。最简单的组织结构是只有两种类型的机器：

(1) 客户机，只含有实现(部分)用户接口的程序。

(2) 服务器，包含了其余的部分，即实现处理和数据层的程序。

在这种组织结构中，所有事情都由服务器来处理，客户基本上只是一个哑终端，也可能有一些漂亮的图形接口。还有很多其他可能性，本节我们将探讨一些更复杂的。

组织客户和服务器的一种方法是，把上一节所介绍的应用层中的程序分布在不同的机器中，如图2.5所示(Umar 1997和Jing等1999)。第一步，让我们来区分只有两种机器(即客户机和服务器)的不同，这就是(**物理**)**两层体系结构**((physically) two-tiered architecture)。

一种可能的组织结构是，只有与终端有关的用户接口部分位于客户机器上，如图2.5(a)所示，由应用程序远程控制数据的显示。另一种是把整个用户接口软件放在客户端，如图2.5(b)所示。在这种情况下，我们基本上是把应用程序分成了一个图形前端，它通过一

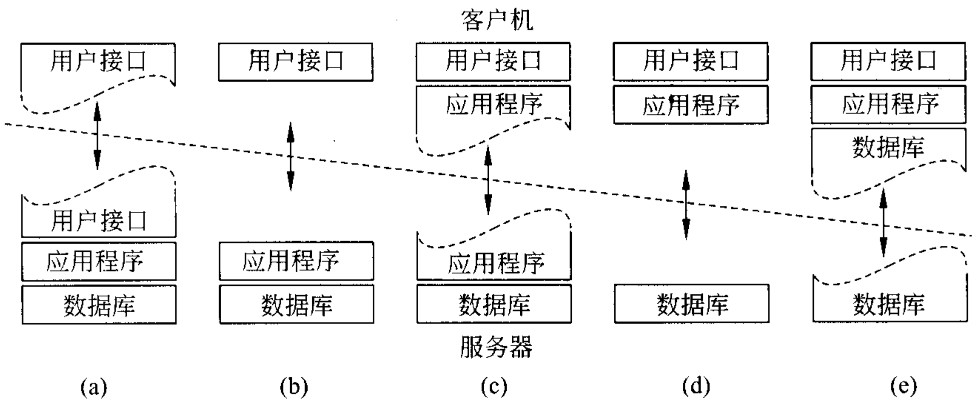


图 2.5 各种客户-服务器组织结构(a)～(e)

个特定应用的协议与应用程序(位于服务器上)的其余部分进行通信。在这种模型中,前端(客户软件)只不过是进行了用于展示应用程序接口的必须处理。

按照这种推理思路继续下去,我们还可以把部分应用程序移到前端中,如图 2.5(c)所示。一个有意义的示例是,应用程序利用一种方式,需要在被处理之前被完全填满。前端可以检查这种方式的正确性和一致性,必要时还可以与用户进行交互。图 2.5(c)的组织结构的另一个示例时文字处理软件,其基本的编辑功能运行在客户端,它们对本地缓冲或内存数据进行操作,但诸如拼写和语法检查等高级支持工具则运行在服务器端。

在很多客户-服务器环境中,图 2.5(d)和 2.5(e)所示的组织结构很普遍。这些组织结构使用在 PC 或工作站的客户机中,这些客户机通过网络连接到分布式文件系统或数据库。大多数的应用程序基本上是运行在客户机上,但所有对文件或数据库项目的操作都是在服务器上进行。例如,很多银行应用程序就是运行在终端用户机上,用户在这里准备事务处理等。一旦完成,应用程序就会与银行服务器上的数据库联系,然后上载事务处理以供以后处理使用。图 2.5(e)表示了客户的本地硬盘含有部分数据的情况。例如,当浏览 Web 网站时,客户端就会慢慢地在本地硬盘构建一个巨大的缓存,其中存储了最近浏览过的 Web 网页。

我们注意到,近些年来,已经有这样一个强大的趋势:不再是如图 2.5(d)和 2.5(e)所示的把客户端软件放在终端用户机上了。在这些情况下,大多数的处理操作和数据存储在服务器端完成。其原因很简单:尽管客户机可以做不少工作,但它们问题也更多,难以管理。把更多的功能放在客户机上,使得客户端软件更容易出错,且更依赖于客户端的底层平台(如操作系统和资源)。从系统管理的角度来说,使用**胖客户**(fat client)并不是最优的。相反,使用如图 2.5(a)～(c)的**瘦客户**(thin client)配置更容易,可以使得用户接口更不那么复杂,获得客户可察觉的性能。

注意,这种趋势并不是意味着我们不再需要分布式系统了。相反,我们可以看到,随着单个服务器被运行在不同机器上的多个服务器取代,服务器端的解决方法正变得越来越是分布式的了。特别是,在区分客户和服务器时,我们忽视了这样一点:有时候服务器可能需要起到一个客户的作用,如图 2.6 所示,这就出现了(**物理**)**三层体系结构**((physically) three-tiered architecture)。

在这种体系结构中,组成部分处理层的程序驻留在单独的服务器上,但也可能部分地分

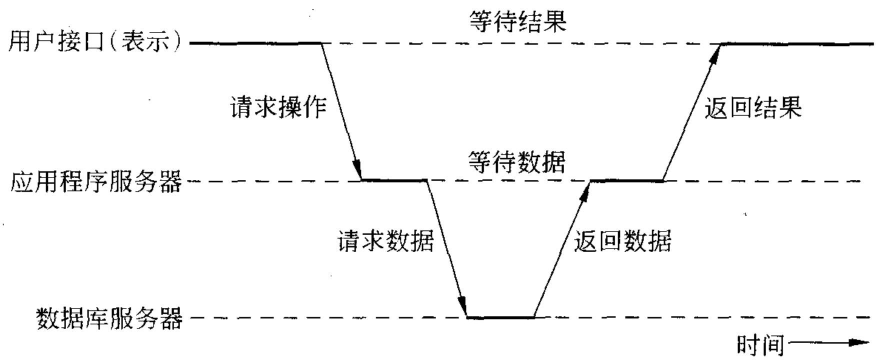


图 2.6 服务器作为一个客户的示例

布在客户和服务器上。三层体系结构的一个典型示例是事务处理。诸如我们在第 1 章所介绍的那样，名为事务处理监视器的单独进程负责协调所有事务，这些事务可能分布在不同的数据服务器上。

另一个很不一般的示例是，Web 网站的组织结构也是一个三层体系结构。在这种情况下，Web 服务器作为一个站点入口点，把请求发送给应用程序服务器，在服务器中进行真正的处理。而应用程序服务器与数据库服务器交互。例如，应用程序服务器可能还负责运行检查某种物品的库存的代码，就像电子书店所提供的那样。要这样做，就需要与含有原始库存数据的数据库交互。在第 12 章我们将讨论 Web 站点的组织结构。

2.2.2 非集中式体系结构

多层客户-服务器把应用程序分成用户接口、处理组件和数据层。不同层直接对应应用程序的逻辑组织结构。在很多商业环境中，分布式处理就等同于把客户-服务器应用程序组成一个多层体系结构。我们把这种类型的分布称为**垂直分布性**(vertical distribution)。垂直分布性的特征是，它是通过按逻辑把不同组件放在不同机器上来获得的。该术语与分布式关系数据库中所使用的垂直分割(vertical fragmentation)概念相关。在分布式关系数据库中，垂直分割就是把表按列分割，从而可分布在多台机器上(Oszu 和 Valduriez 1999)。

同样，从系统管理的角度来说，具有垂直分布性有助于：功能可以从逻辑上和物理上分割在多台机器上，每台机器按特定的功能组进行定制。然而，垂直分布性只是组织客户-服务器应用程序的一种方法。在现代体系结构中经常包含了客户和服务器的分布性。我们称之为**水平分布性**(horizontal distribution)。在这种类型的分布式系统中，客户或服务器可能在物理上被分割成逻辑上相等的几个部分，但每个部分都操作在整个数据集中自己共享的部分，从而实现负载的平衡。本节我们将介绍一种现代的、能够支持水平分布性的系统体系结构，称为**点对点系统**(peer-peer system)。

从高层的角度来看，构成点对点系统的进程是全部相等的。这意味着，需要执行的功能由构成分布式系统的每个进程来表示。因此，进程间的很多交互是对称的：每个进程作为客户的同时又是服务器(这又称为**服务器客户**(servent))。

有了这种对称行为，点对点体系结构就绕过了覆盖网络(overlay network)中如何组织进程的问题，也就是说，网络中的结点是由进程和表示各种可能的通信通道的链接(通常实

现为 TCP 连接)形成的。通常,一个进程不能与另外任意一个进程直接通信,而是要求通过一个可用的通信通道来发送消息。现在有两种类型的覆盖网络:一种是结构化的,另一种是非结构化的。这两种类型在(Lua 等 2005)中进行了扩展介绍,并给出了大量示例。Aberer 等(2005)给出了一种参考体系结构,对不同类型的点对点系统做了更正式的对比。

2.2.2.1 结构化的点对点体系结构

在结构化的点对点体系结构中,覆盖网络是用一个确定性的过程来构成的。这个使用最多的进程是通过一个**分布式哈希表**(distributed hash table,DHT)来组织进程的。在基于 DHT 的系统中,从一个大的标识符空间中选取一个随机关键值赋给数据项,例如 128 位或 160 位的标识符。同样,从这个标识符空间中选取一个随机数赋给该系统中的结点。这样,每个基于 DHT 系统的核心是实现一种高效且确定的方案,根据某种距离尺度把数据项的关键值唯一地映射给结点的标识符(Balakrishan 2003)。最重要的是,当查找数据项时,会返回对应数据项的结点的网络地址。这可以通过把数据项的请求路由给相应的结点来完成。

例如,在 Chord 系统(Stoica 等 2003)中,结点按逻辑组成一个环,这样,关键值 k 的数据项映射到最新标识符 id≥k 的结点。该结点称为关键值 k 的后继者,记作 succ(k),如图 2.7 所示。要真正查找数据项,运行在任意结点上的应用程序可调用函数 LOOKUP(k),它可返回 succ(k)的网络地址。此时,应用程序就可以与该结点联系以获得该数据项的副本。

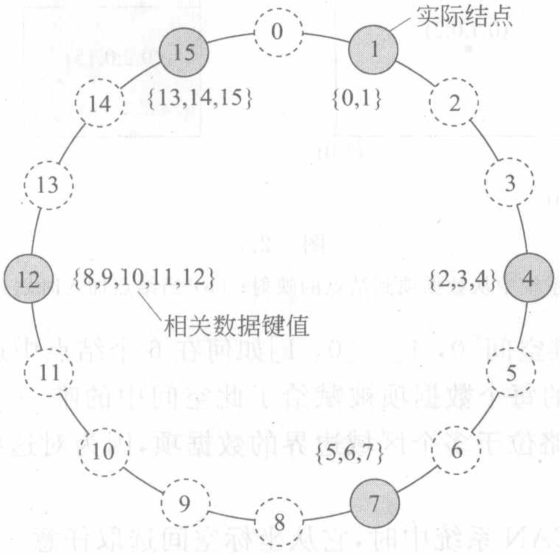


图 2.7 在 Chord 系统中,从数据项到结点的映射

这里我们不准备深入讨论查找关键值的算法,而是等到第 5 章再讨论。在第 5 章我们将介绍各种命名系统的详细内容。这里我们重点关注结点如何把它们自己组织成一个覆盖网络,即**成员管理**(membership management)。认识到结点的查找并不按照图 2.7 所示的环中结点的逻辑组织结构很重要。每个结点有维护到其他结点的捷径,这样,查找工作可以在 O(log(N))步内完成,其中 N 是覆盖网络中的结点数。

现在再来看看 Chord 系统。如果某个结点要加入该系统,它首先生成一个随机标识符 id。注意,如果标识符空间足够大,假设随机数生成器的质量比较好,那么,所生成的标识符与已赋给实际结点的标识符相等的概率就接近于零。然后,结点就可以进行一个对 id 的查

找，返回网络地址 succ(id)。此时，所加入的结点只需与 succ(id) 及其前继者联系，并把自己插入到该环中即可。当然，这种方案需要每个结点都存储了其前继者的信息。插入操作还将使得 succ(id) 转移到其关键值与结点 id 相关联的每个数据项中。

结点离开也很简单：结点 id 告知其前继者和后继者它要离开，并把其数据项转移给 succ(id)。

在其他基于 DHT 的系统中也可以遵循类似的方法。例如，我们来看看(Ratnasamy 等 2001)所介绍的上下文可编址网络(Content Addressable Network，CAN)。CAN 部署一个 d 维的笛卡儿坐标空间，并按参与该系统的所有结点进行划分。为了便于说明，让我们来看看只有二维的情况，如图 2.8 所示就是这样一个示例。

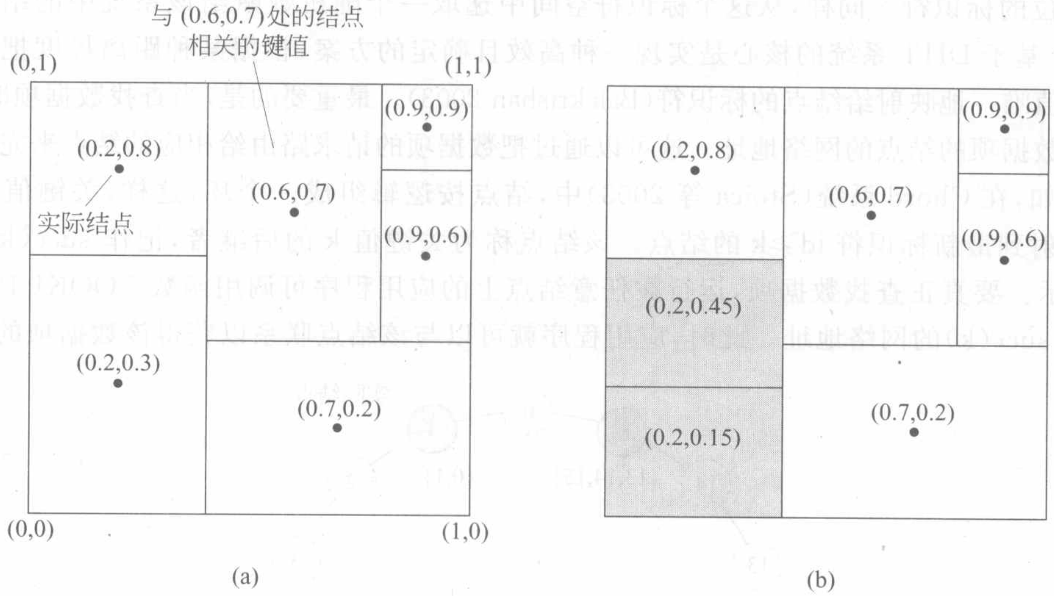


图 2.8

(a) CAN 系统中从数据项到结点的映射；(b) 当结点加入时划分一个区域

图 2.8(a) 显示了二维空间[0，1]×[0，1]如何在 6 个结点中进行划分。每个结点都有一个相关的区域。CAN 的每个数据项被赋给了此空间中的唯一一点，这样，哪个结点负责哪个数据也就清楚了(忽略位于多个区域边界的数据项，因为对这些数据项使用了一个确切赋值)。

当结点 P 要加入到 CAN 系统中时，它从坐标空间选取任意一点，然后在它所位于的区域中查找结点 Q。该查找工作通过基于位置的路由来完成，详细内容推迟到后面章节中介绍。然后，结点 Q 把其区域分成两个等份，如图 2.8(b) 所示，其中一半赋给了结点 P。每个结点都与邻结点保持联系。当分割一个区域时，通过询问结点 Q，所加入的结点 P 就可以很容易地知道哪个是其新的邻结点。在 Chord 系统中，结点 Q 的数据项转移给了结点 P。

在 CAN 中，结点的离开就稍微困难些。假设在图 2.8 中坐标为(0.6，0.7)的结点要离开。其区域将赋给它的一个邻结点，即位于(0.9，0.9)的结点，但很显然，这并不能融合成一个矩形。在这种情况下，位于(0.9，0.9)的结点将接管这个区域并通知旧结点。显然，这可能导致坐标空间的不对称划分，因此，要定期地启动一个后台进程来重新划分整个空间。

2.2.2.2 非结构化的点对点体系结构

非结构化的点对点体系结构主要依靠随机化的算法来构造覆盖网络。其主要思想是，每个结点维护一个邻结点列表，但这个列表是以一种或多或少为随机的方法来构造的。同样，数据项也是假定随机地放在结点中的。因此，当结点要定位某个特定的数据项时，它所能有效地做的唯一事情就是用一个查找查询来泛洪该网络（Risson 和 Moors 2006）。在第 5 章我们将回过头来讨论在非结构化覆盖网络中的查找，这里只关注成员管理。

很多非结构化的点对点系统的一个目标就是构造一个类似于**随机图**（random graph）的覆盖网络。其基本模型是每个结点都维护一个含有 c 个邻结点的列表，在理想情况下，每个这种邻结点表示的是从当前结点集中随机选择出来的“活”结点。这个邻结点列表又称为**部分视图**（partial view）构造这种部分试图的方法有很多。（Jelasity 等 2004，2005a）开发了一个框架，吸收了很多不同的覆盖构造算法，用于进行评估和比较。在这个框架中，假定结点是有规则地从它们的部分视图交换项目的。每个项目标识网络中的另一个结点，且有一个相关的年龄，表明引用该结点的时间长短。这里使用了两个线程，如图 2.9 所示。

主动线程发起与另一个结点的通信。它从其部分视图中选择结点。假设这些项目需要被“推”到所选择的对等结点，那么它就可以连续地构造一个含有 c/2＋1 个项目（包括标识其自身的一个项目）的缓冲区。其他项目是取自当前部分视图的。

如果结点也是“拉”模式，那么它将等待来自所选择的对等结点的响应。同时，该对等结点也通过被动线程构造一个缓冲区，如图 2.9(b)所示，其活动非常类似主动线程。

这里的关键点是新部分视图的构造。对发起线程和对等线程，该视图都是正好有 c 个项目，其中有一部分是来自所接收的缓冲区的。这里基本上有两种构造新视图的方法。首先，这两个结点要决定丢弃掉那些已发送给对方的项目。这就意味着它们将“抹”去部分的原始视图。第二种方法是丢弃尽可能多的“旧”项目。通常，这两种方法是互补的，详细内容请参见（Jelasity 等 2005a）。实践证明，用于构造覆盖的很多成员管理协议适合这种框架。这里有很多有趣的因素。

```
主动线程的动作（周期性重复）：
从当前的部分视图中选择一个对等体 P
if PUSH_MODE{
    mybuffer=[(MyAddress,0)];
    permute partial view;
    move H oldest entries to the end;
    append first c/2 entries to mybuffer;
    send mybuffer to P;
} else {
    send trigger to P;
}
if PULL_MODE {
    receive P's buffer;
}
从当前视图和 P 的缓冲区构建一个新的部分视图；
增加新的部分视图中每个项目的年龄
```

(a)

```
被动线程的动作：
从任意进程 Q 中接收缓冲
if PULL_MODE{
    mybuffer=[(MyAddress,0)];
    permute partial view;
    move H oldest entries to the end;
    append first c/2 entries to mybuffer;
    send mybuffer to P;
}
从当前视图和 P 的缓冲区构建一个新的部分视图；
增加新的部分视图中每个项目的年龄
```

(b)

图 2.9
(a) 主动线程所采用的步骤；
(b) 被动线程所采用的步骤

首先，我们假设，当一个结点加入时，它与任意其他结点联系，该结点可能是来自一个已知的访问点列表。除非我们假设访问点是高度可用的，否则它就只是覆盖网络的一个常规成员。在这种情况下，事实证明，那些只使用推模式或拉模式的协议很容易导致覆盖网络的断开。换句话说，结点组将变成孤立的，再也

无法达到网络中的其他结点。显然，这是一种不受欢迎的特性，出于这个原因，让结点真正“互换”项目是更有意义的。

第二，如果结点按照常规的规则互换部分视图，离开网络是一个非常简单的操作。在这种情况下，结点无需通知其他结点就可以离开。这里所发生的是，当结点 P 选择了一个邻结点，假设为结点 Q，发现结点 Q 不再响应了，那么它就可以从其部分视图中删除该项目，以便选择另外一个对等结点。事实证明，当构造一个新的部分视图时，结点遵循的是丢弃尽可能多的旧项目的策略，已离开的结点将很快被遗忘。换句话说，指向已离开的结点的项目将自动快速地从部分视图中删去。

但是，遵循这种策略是要付出代价的。为了便于解释，我们来看看结点 P 和一个结点集，在这个结点集的部分视图中有指向 P 的一个项目。从技术上来说，这称为结点的**内阶**(indegree)。结点 P 的内阶越高，其他结点与其联系的可能性就越大。换句话说，P 就越容易成为普遍结点，这也就更容易使其成为负载不平衡点。从系统的角度来说，丢弃旧项目会使得结点的内阶提高。还有其他一些要权衡的，详细内容请参见(Jelasity 等 2005a)。

2.2.2.3 覆盖网络的拓扑管理

尽管看上去结构化和非结构化的点对点系统形成了严格独立的类型，但实际上并不需要这样(Castro 等 2005)。一个关键的因素是，通过仔细地交换和选择部分视图，旧可以构造和维护覆盖网络的特殊拓扑结构。采用一种两层的方法就可实现拓扑管理，如图 2.10 所示。

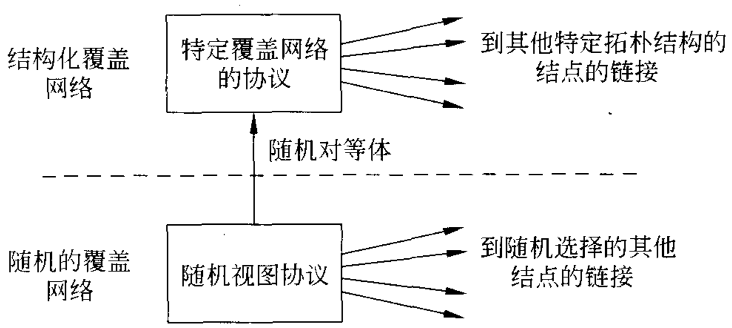


图 2.10 使用非结构化的点对点系统的技术构造和维护特殊覆盖拓扑的两层方法

最底层构成了一个非结构化的点对点系统，其中的结点定期地交换部分视图中的项目，目的是维护一个精确的随机视图。这里的精确性指的是，应该用指向随机选择的“活的”结点的项目来填充部分视图。

最底层把其部分视图传递给更高层，在此进行另外的项目选择。这就产生了对应于所希望的拓扑结构的邻结点的第二个列表。(Jelasity 和 Babaoglu 2005)提出使用一个“排名函数”，利用该函数，根据与给定结点相关的某些标准，对结点进行排序。一个简单的排名函数是按照与给定结点 P 的距离对结点集排序。在这种情况下，只要最低层不断地传递随机选择的结点，结点 P 就可以逐步创建最近邻结点列表。

下面举例进行说明，看看一个大小为 N×N 的逻辑网络，网络的每个点上放置一个结点。要求每个结点维护一个含有 c 个最近邻结点的列表，其中位于(a_1, a_2)和(b_1, b_2)的两个结点之间的距离定义为 d_1+d_2，且 $d_i=\min(N-|a_i-b_i|, |a_i-b_i|)$。如果最底层定期地

运行如图 2.9 所示的协议，那么该拓扑就可以发展成如图 2.11 所示的花托形状。

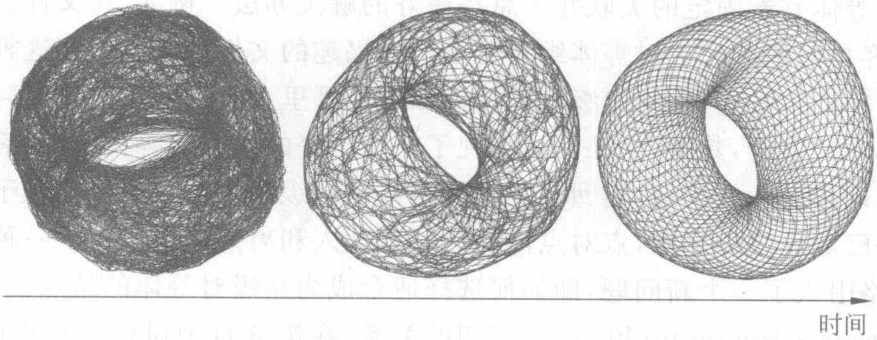


图 2.11 使用一个两层非结构化的点对点系统生成一个特定的覆盖网络

当然，也可以使用完全不同的排名函数。特别令人感兴趣的是与捕捉存储在对等结点的数据项的**语义近似性**(semantic proximity)有关的那些函数。这种近似性可以构造语义覆盖网络，可以用于非结构化点对点系统的高效查找算法。在第 5 章讨论基于属性的命名时再介绍这些系统。

2.2.2.4 超级对等体

在非结构化的点对点系统中，随着网络的增大，相关数据项的定位会变得很困难。产生这种可扩展性问题原因很简单：不存在一种确定的方法把查找请求路由到特定的数据项，结点所能借助的唯一技术是泛洪该请求。泛洪的危害很多，我们将在第 5 章中讨论，作为一种替代方法，很多点对点系统提出了使用能够维护数据项索引的特别结点。

在其他情况下，放弃点对点系统的对称性也是明智之举。我们来看看相互提供资源的结点间协作。例如，在协作的**上下文传送网络**(content delivery network，CDN)中，结点可以为驻留 Web 网页提供存储空间，让 Web 客户能访问最近的页面，从而可以快速地访问。此时，结点 P 可能需要查寻网络特定部分的资源。在这种情况下，使用能收集一些结点的资源使用情况的**代理程序**(broker)就可以快速地选择具有足够资源的结点。

这些能维护一个索引或充当一个代理程序的结点通常称为**超级对等体**(superpeer)。正如其名字所表示的那样，超级对等体经常也是组织在点对点系统网络中，形成一个分层组织结构，详细介绍请参见(Yang 和 Garcia-Molina 2003)。这种组织结构的一个简单示例如图 2.12 所示。在这种组织结构中，每个常规的对等体作为一个客户连接到超级对等体。所有往来于常规对等体的通信都是先通过与该对等体关联的超级对等体的。

在很多情况下，客户-超级对等体的关系是固定的：一个对等体只要加入到网络中，它就只归依于某个超级对等体，并一直保持不变，直到它离开该网络。显然，超级对等体应该是生存期较长的进程，而且具有较高的可用性。为了弥补超级对等体潜在的不稳定性，可以部署备用方案，如把每个超级对等体

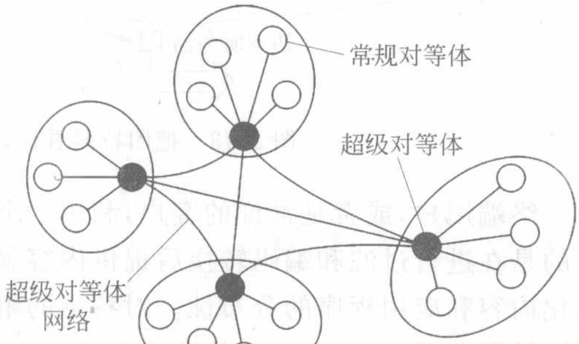


图 2.12 超级对等体网络中结点的分层组织结构

与另一个配对，并要求客户与这两个超级对等体同时连接。

与超级对等体有着固定的关联并不总是最好的解决办法。例如，在文件共享网络中，更好的办法是，客户归依的超级对等体维护着客户感兴趣的文件的索引。在这种情况下，客户查找特定文件的机会就更大，其超级对等体也知道到哪里去寻找。(Garbacki 等 2005)描述了一个相对简单的方案，其中，如果客户发现了一个更好的超级对等体，那么客户-超级对等体的关系就可以改变。特别地，返回查找操作结果的超级对等体表明了它优于其他对等体。

正如我们已经见过的那样，点对点网络为结点加入和离开网络提供了一种灵活的方法。然而，超级网络引入了一个新问题，即如何选择适合成为超级对等体的结点。该问题与领导者选举问题(leader-election problem)有密切的关系，在第 6 章中讨论点对点网络中的超级对等体选择时将讨论这个问题。

2.2.3 混合体系结构

到目前止，我们主要关注的是客户-服务器体系结构和一些点对点体系结构。很多分布式系统综合了多种体系结构的特性，如我们刚刚介绍的超级对等体网络。本章我们将介绍一些特殊类型的分布式系统，这些系统把客户-服务器体系结构和非集中式体系结构组合在了一起。

2.2.3.1 边界服务器系统

按混合体系结构组织成的一种重要的分布式系统是**边界服务器系统**(edge-server system)。这种系统部署在因特网中，其中，服务器放置在网络的“边界”。这种边界是由企业网络和实际的因特网之间的分界线形成的。例如，**因特网服务提供商**(Internet Service Provider，ISP)。家庭终端用户通过 ISP 连接到因特网，该 ISP 就可以认为是因特网的边界。这就形成了如图 2.13 所示的一般组织结构。

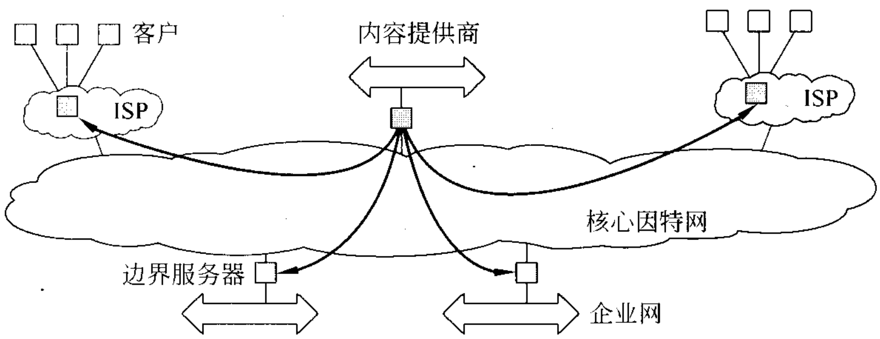


图 2.13 把因特网看作是由一系列边界服务器组成的

终端用户，或者是常说的客户，利用一个边界服务器连接到因特网。边界服务器的主要目的是在进行过滤和编码转换后提供内容服务。更令人感兴趣的是边界服务器集可以用于优化内容和应用程序的分布性。对特定的组织结构的基本模型是，边界服务器可用作为一个初始服务器，所有内容都从这里开始。这种服务器可以使用其他边界服务器来复制 Web 页面或其他(Leff 等 2004；Nayate 等 2004；Rabinovich 和 Spatscheck 2002)。在第 12 章我们将回过头来介绍边界服务器系统。

2.2.3.2 协作分布式系统

混合结构主要部署在协作分布式系统中。在很多这样的系统中，主要问题是首先启动起来，因为经常会部署一个传递的客户-服务器结构。一旦某个结点要加入系统，就可以使用完全非集中式的组织结构，以用于协作。

为了使事情更具体，我们首先来看看文件共享系统 BitTorrent (Cohen 2003)。BitTorrent 是一种点对点文件下载系统。其工作原理如图 2.14 所示。基本思想是，当一个终端用户要查找某个文件时，他可以从其他用户那里下载文件块，直到所下载的文件块能够组装成完整的文件为止。一个重要的设计目标是确保协作性。在大多数文件共享系统中，参与者只是下载文件，其他什么事情也不做(Adar 和 Huberman 2000; Saroiu 等 2003; Yang 等 2005)。总之，只有当下载客户为他人提供了内容，文件才可被下载。我们马上会回来讨论这种行为。

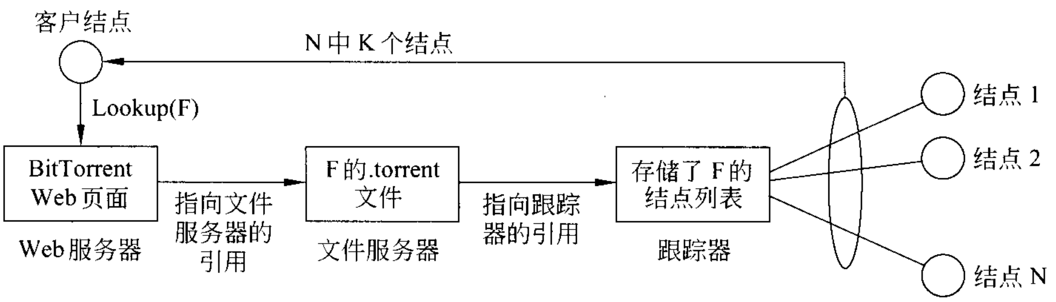


图 2.14 BitTorrent 的工作原理

要下载一个文件，用户需要访问一个全局目录，它只是一些知名 Web 站点中的一个。这种目录含有指向名为.torrent 文件的引用。一个.torrent 文件含有要下载特定文件的信息。特别地，它指向一个**跟踪器**(tracker)。这是一个服务器，它保持有活动结点的精确记录，而这些结点具有所请求的文件块。一个活动结点就是当前正在下载另一个文件的结点。显然，这将会有很多不同的跟踪器，但通常每个文件(或文件集)只有一个跟踪器。

一旦结点确定了从哪里可以下载文件块，下载结点就变成了活动的。此时，它就被强制为其他结点提供帮助，例如，提供它正在下载但其他结点还没有的文件块。这种强制性来自于一个非常简单规则：如果结点 P 知道结点 Q 正在下载而不只是上载，P 就可以决定降低发送数据给 Q 的速率。只要 P 需要从 Q 那里下载，这种方案就能很好工作。鉴于这个原因，结点往往有很多指向其他结点的引用，使得它们处于一个更好的位置来交换数据。

显然，BitTorrent 综合了集中式和非集中式两种方法。结果证明，该系统的瓶颈果然是由跟踪器导致的。

另一个例子是 Globule 协作内容分布式网络(Pierre 和 van Steen 2006)。Globule 非常像前面介绍的边界服务器体系结构。在这里，没有边界服务器，终端用户(和组织结构)自愿提供增强的 Web 服务器，它们在 Web 页面的复制中可以协同工作。在最简单的形式中，每个这样的服务器具有以下组件：

(1) 可以把客户请求重定向到其他服务器的组件。

(2) 用于分析访问模式的组件。

(3) 用于管理Web页面复制的组件。

例如，如果由Alice提供的服务器是Web服务器，它通常负责处理Alice的Web站点的数据流量，称之为这个站点的**初始服务器**(origin server)。它可以与其他服务器协作，例如，由Bob提供的服务器，把来自Bob站点的页面驻留。此时，Globule是一种非集中式的分布式系统。对Alice的Web站点的请求被转发给她的服务器，在这里，这些请求又可以重定向到其他服务器。这种系统还支持分布式重定向。

但是，Globule也有一个集中式组件，即**代理程序**(broker)。代理程序负责注册服务器，使这些服务器能被其他服务器所认识。服务器与代理程序之间的通信类似于客户-服务器系统中的通信。出于可用性的考虑，该代理程序可以被复制，我们后面将介绍，这种类型的复制被广泛应用，目的是获得可靠的客户-服务器计算。

2.3 体系结构与中间件

当我们讨论体系结构问题是，出现的一个疑问是中间件位于何处。正如我们在第1章中讨论的那样，中间件在应用程序和分布式平台之间形成了一个层，如图1.1所示。一个重要的目的是提供一定程度的透明性，也就是一定程度上向应用程序隐藏数据处理和控制的分布性。

在实践中常见的中间件系统实际上是遵循一定的体系结构风格的。例如，很多中间件解决方法采用的是基于对象的体系结构，如CORBA(OMG 2004a)。而像TIC/Rendzvous(TIBCO 2005)之类的其他中间件则遵循基于事件的体系结构。在后面的章节中，我们将遇到更多的体系结构示例。

使中间件按照特定的体系结构融合到系统中，有利于使应用程序的设计变得更简单。但是，一个明显的缺陷是，中间件可能不是应用程序开发人员脑海中的最佳选择。例如，CORBA最初只提供对象，这些对象可以被远程客户激活。后来，人们感觉到只有这种交互形式很受限制，因此又加入了诸如消息传递之类的其他交互模式。显然，添加新特性容易提升中间件的解决方法。

另外，尽管中间件意味着能提供分布式透明性，人们通常觉得，特定的解决方法应该能适应应用程序的需求。这个问题的一个解决方法是，实现中间件的多个版本，每个版本按照特定类型的应用程序进行裁剪。通常认为一种更好的方法是，使中间件系统按照应用程序的需要能很容易地配置、适应和定制。因此，现在系统的开发在策略与机制之间有更严格的分离。这就产生了多种机制，利用这些机制，可以修改中间件的行为(Sadjadi 和 McKinley 2003)。让我们来看看最常用的一些方法。

2.3.1 中断器

从概念上来说，中断器只不过是一种软件结构，它能中断正常的控制流，从而允许其他(应用程序特定的)代码运行。要使得中断器通用，要求相当的实现努力，详见(Schmidt 等 2000)，而且不清楚这种通用性是否就优于严格的适应性和简单性。同样，在很多情况下，只

具有有限的中断能力从整体上能提高软件和分布式系统的管理。

为了使事情更具体，我们来看看在很多基于对象的分布式系统中获得支持的中断。其基本思想很简单：对象 A 可以调用属于对象 B 的方法，而对象 B 与对象 A 位于不同的机器之中。本书后面将详细解释，这种远程对象调用是以如下三个步骤来完成的：

(1) 对象 A 提供一个本地接口，该接口与对象 B 提供的接口完全相同。A 只需调用该接口中的一个方法变量即可。

(2) A 的调用转换为一个通用的对象激活，这可以通过由与 A 在同一机器的中间件所提供的对象激活接口来完成。

(3) 最后，该通用对象激活转换为一个消息，该消息可以通过由 A 的本地操作系统所提供的传输层网络接口来发送。

这种方案如图 2.15 所示。

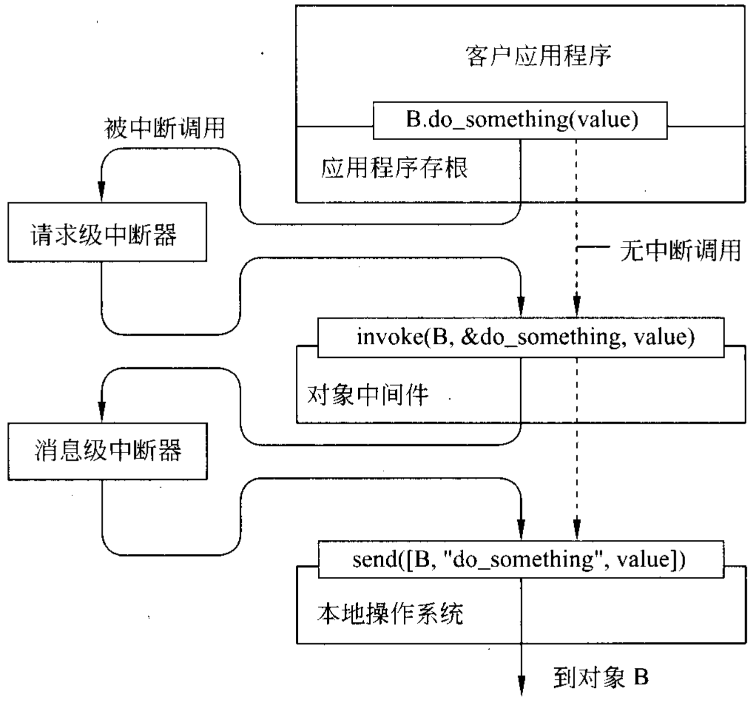


图 2.15　使用中断器来处理远程对象激活

第一步之后，B. some_something(value)调用转换成诸如 invoke(B, &do_something, value)之类的通用调用，它具有一个指向 B 的方法的引用，以及与该调用一起的参数。现在假想对象 B 被复制了。此时，每个副本都必须真正地被激活。很明显这是中断能起作用的地方。**请求级中断器**(request-level interceptor)将做的只是为每个副本调用 invoke(B, &do_something, value)。该调用的优美之处是对象 A 不需要知道 B 的副本，而且中间件也不必须要有用于处理该复制调用的特别组件。只有请求级中断器才需要知道 B 的副本，该中断器可能被加入到中间件中。

最后，远程对象的调用必须在网络上发送。实际上，这就意味着由本地操作系统提供的消息传递接口需要被激活。在这一级中，**消息级中断器**(message-level interceptor)可能协助把该激活转移给目标对象。例如，假想参数 value 真的是对应着一个巨大的数据数组。在这种情况下，明智的做法是把该数据数组拆分成多个更小的部分，然后，在目的地再把它

组装在一起。这种拆分可提高性能和可靠性。同样，中间件并不需要知道这种拆分的细节，低层的中断器可以透明地处理与本地操作系统通信的其他工作。

2.3.2 自适应软件的常见方法

实际上，中断器所提供的是适应中间件的一种方法。自适应的需要源自于这样一个事实：分布式应用程序所运行的环境是不断变化的。这些变化包括移动性导致的结果，网络服务质量的巨大差别，硬件故障，电源消耗等等。对这些变化的相应处理不是由应用程序来负责，而是把该工作放在中间件中。

来自环境的强烈影响促使很多中间件设计人员考虑使用自适应软件结构。但是，自适应软件并不如所期望的那样成功。由于很多研究人员和开发人员认为它是现代分布式系统的一个重要方面，因此让我们来简要地看看它。McKinley 等(2004)把实现软件自适应的基本技术分成以下三种：

(1) 要点分离；

(2) 计算映像；

(3) 基于组件的设计。

要点分离与模块化系统的传统方法有关：把实现功能的部分与负责诸如可靠性、性能、安全等其他事情(称为额外功能)的部分分开。有人认为，为分布式应用程序开发的中间件大部分是处理与应用程序无关的额外功能。这里的主要问题是，利用模块化的方法把这些额外功能分开并不容易。例如，简单地把安全性放在一个单独的模板中并不能工作。同样，你也无法想象能把容错性隔离在一个单独的盒子之中，作为一个独立的服务来销售。分离并且进而把这些要点编织到(分布式)系统是**面向局部软件开发**(aspect-oriented software development)所解决的主要题目(Filman 等 2005)。然而，面向局部的方法还未能成功地应用于开发大型分布式系统，要达到这个层面，还有很长的路要走。

计算映像指的是程序检查自己，以及如果有必要，调整其行为的能力(Kon 等 2002)。计算映像已置入到程序设计语言(包括 Java)之中了，为运行时调整提供了一个强大的工具。另外，一些中间件系统提供了应用映像技术的方法。但是，正如面向局部软件开发方法一样，计算映像中间件还未能证明自己是管理大型分布式系统的复杂性的有力工具。正如(Blair 等 2004)所介绍的那样，把计算映像应用到应用程序的更广泛领域还有待时日。

最后，基于组件的设计是通过组件的不同组合来支持自适应的。系统可以是在设计时静态地配置的，或者是在运行时动态地配置的。后者要求支持后绑定。后绑定是一种已成功应用于程序设计语义环境的技术，而且对操作系统来说，模块可以随意加载和卸载。现在的研究能在运行时自动选择组件的最好实现了(Yellin 2003)，但同样，对分布式系统来说，进程仍很复杂，尤其是要考虑到一个组件的替换要求知道替换其他组件将会产生什么结果。在很多情况下，组件并没有人们认为的那样多独立性。

2.3.3 讨论

分布式系统的软件体系结构，主要以中间件为基础，是庞大而复杂的。在很大程度上，

这种庞大和复杂性是来自于提供分布式透明性的要求的。同时，应用程序具有特定的额外功能要求，这与完全实现透明性是相冲突的。这种通用性和特殊性需求之间的冲突，使得中间件应是高度灵活的。但是，付出的代价是复杂性。例如，据(Zhang 和 Jacobsen 2004)报道，特定软件产品从它引入后仅4年时间，其大小就增长了50%，而在相同时间内，这种产品的文件总数增长了两倍。显然，这并不是我们所追去的方向。

考虑到现在所有大型软件系统都要求运行在网络环境下，我们可以问问自己，分布式系统的复杂性是否是试图取得分布式透明性固有的特性？当然，诸如开放性之类的问题也同样重要，但对灵活性的需要从未有像中间件那样普遍。

(Coyler 等 2003)认为所需的是应更加关注外部简单性，即利用组件构建中间件的更简单方法，以及应用程序的独立性。上面所介绍的技术的解决方案都值得探讨。特别地，到目前为止所提出的技术都没有被大范围地采用，它们也没有被成功地应用到大型系统的记录。

这里的基本假设是，我们所需要的自适应软件是软件应可以随着环境的变化而改变。但是，你应该问问，适应变化环境是否是改变软件的很好借口。硬件故障、安全攻击、能量消耗等看上去都是环境影响因素，都是软件所期望解决的。

支持自适应软件最有力、当然也是最有效的理由是，很多分布式系统是不能停机的。这个限制要求有替代和提升组件的解决方案，但不清楚上面提出的解决方案是否是处理这种可持续问题的最好办法。

于是，剩余的问题是分布式系统应能对环境的变化做出反应，例如，通过更换分配资源的策略等。所有能启用自适应的软件组件都应处在合适的位置。包含在这些组件中的算法和行为指令能改变其位置。这里的困难是让这种响应行为在无人干预的情况下发生。例如，在讨论分布式系统的物理组织结构时，这种方法能工作得更好。接下来我们讨论这种系统体系结构。

2.4 分布式系统的自我管理

分布式系统，主要是与它们相关的中间件，需要提供常用的解决方案来屏蔽掉那些不希望有的网络固有特性，这样它们就可以支持尽可能多的应用程序。另一方面，完全透明性并不是大多数应用程序所真正需要的，它们还需要支持特定应用程序的解决方案。基于这个原因，我们认为，分布式系统应是自适应的，但主要是调整它们的运行行为，而不是软件组件。

当需要自适应自动完成时，我们可以看到，在系统体系结构与软件体系结构之间有很强的相互作用。另外，我们需要以某种方式来组织分布式系统的组件，使得能实现监视和调整，同时，我们有需要决定处理自适应的进程在何处运行。

本节我们来介绍以高级反馈控制系统的形式来组织分布式系统，它允许自动自适应变化。这种现象称为**自治计算**(autonomic computing)(Kephart 2003)或**自主系统**(self-star system)(Babaoglu 等 2005)。后一个名字表明自适应的多样性：自我管理、自我恢复、自我配置、自我优化等。这里我们用名称自我管理来表示各种自适应性。

2.4.1 反馈控制模型

对自我管理系统有很多不同的观点，但最常用的是假设自适应是通过一种或多种**反馈控制循环**(feedback control loop)来实现的。相应地，以这些循环组织的系统称为**反馈控制系统**(feedback control system)。反馈控制已在各种工程领域应用多年了，其数学基础知识也逐步在计算系统中得到应用(Hellserstein 等 2004；Diao 等 2005)。对自我管理系统，其体系结构问题是最令人感兴趣的。这种组织结构背后的基本思想很简单，如图 2.16 所示。

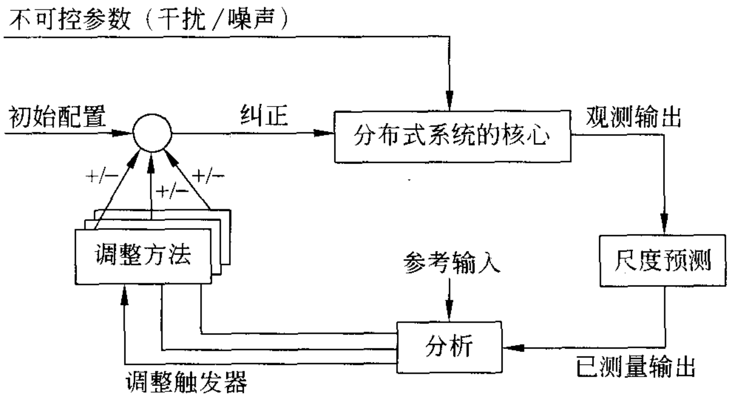


图 2.16　反馈控制系统的逻辑组织结构

反馈控制系统的核心由需要管理的组件形成。这些组件能通过可控输入参数驱动，但它们的行为会受所有不可控输入(又称为干扰或噪声输入)影响。尽管这些干扰往往是来自于分布式系统的运行环境，但也可能是没有预计到的组件间交互所引起的无法预料的行为。

形成反馈控制循环有三个元素。首先，系统本身需要被监视，这要求对系统的各个方面进行测量。在很多情况下，测量行为是说得容易做到难。例如，在因特网中的往返延时差别很多，而且还与要测量的确切内容有关。在这种情况下，要精确地测量一个延时真的很困难。如果结点 A 需要估计两个完全不同的结点 B 和结点 C 之间的延时，且不介入这两个结点，事情就更加复杂了。由于这个原因，控制反馈循环往往包含有一个逻辑**尺度预测组件**(metric estimation component)。

反馈控制循环的另一个部分是分析上述测量值，并把它们与参考值进行比较，这就是反馈分析组件，它是控制循环的核心部分，因为它包含了决定自适应的各种算法。

其他的组件组是由不同的机制组成的，这些组件直接影响着系统的行为。这里可以有多种不同的机制：副本的放置、调度优先权的改变、服务交换、因可用性而移动数据、重定向对不同服务器的请求等。分析组件需要知道这些组件及其对系统行为的(预期)影响。因此，要触发一种或多种机制，以便随后观察其影响。

一个令人感兴趣的现象是反馈控制系统循环也与系统的手动管理相适应。其主要的不同是分析组件由管理人员代替了。但是，为了恰当地管理各种分布式系统，这些管理人员需要合适的监视设备和机制来控制系统的行为。显然，要恰当地分析所测量的数据和触发正确的动作，自我管理系统的开发很困难。

应该强调的是，图 2.16 所示的是自我管理系统的逻辑组织结构，它与我们在讨论软件

体系结构时所见到的软件逻辑组织结构相对应。物理组织结构可能会有很大的不同。例如，分析组件可能是完全分布在这个系统中的。同样，性能测量通常是在组成分布式系统的每台机器中来完成的。下面让我们来看看一些具体的示例是如何以自动的方式进行监视、分析和纠正分布式系统的。这些示例还阐明了逻辑组织结构与物理组织结构之间的差异。

2.4.2　示例：用 Astrolabe 监视系统

我们要考察的第一个示例是 Astrolable(Van Reness 等 2003)，它是一个可以支持特大型分布式系统的监视系统。在自我管理系统的上下文中，Astrolable 被定位为一种可以察看系统行为的常用工具。其输出可输入到分析组件，用于决定正确的动作。

Astrolable 把一个大的主机集组织成分层的区域。最低层区域只由一个主机构成，因而把它们归组为大小可增的区域。最高层区域涵盖了所有主机。每个主机运行一个 Astrolable 进程，称之为代理，它将收集区域的信息。该代理还能与其他代理通信。目的是把区域信息扩散到整个系统。

每个主机维护一个属性集，用来收集本地信息。例如，一个主机可以跟踪它存储的特定文件、其资源使用情况等。只有这些属性是由主机直接维护的，即最低层是可写的。每个区域还有一个属性集，但这些属性是从更低层的值计算而来的。

请看图 2.17 所示的简单示例，它有三个主机 A、B 和 C，它们归组为一个区域。每个主机跟踪其 IP 地址、CPU 负载、可用的空闲内存以及活动的进程数目。每个属性都可用来自每个主机的本地信息直接写入。在区域层，只有聚集信息，如 CPU 负载平均值或活动进程的平均值才被收集。

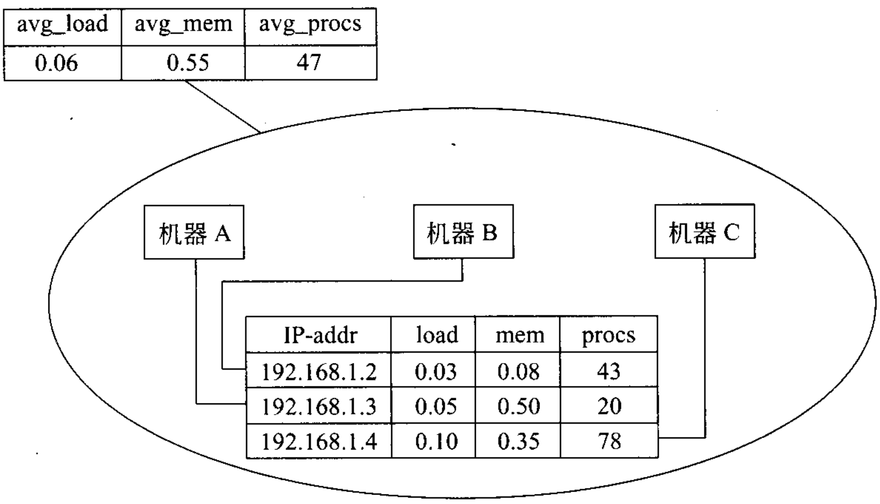


图 2.17　Astrolable 中的数据集合和信息聚集

图 2.17 显示，由每台机器收集起来的信息可以看作是数据库中的一个记录，这些记录共同形成了一个关系(表)。这种表示是 Astrolable 查看所有已收集数据的方法。然而，每个区域的信息只能是从由主机所维护的基本记录计算而来。

聚集信息是通过可编程的聚集函数获得的，这些函数非常类似于关系数据库语言 SQL 中的函数。例如，图 2.17 中的主机信息维护在一个称为 hostinfo 的本地表中，我们就可以

用如下简单的 SQL 查询，来收集包含有主机 A、B 和 C 的区域的平均进程数：

```
SELECT AVG(proc) AS avg_procs FROM hostinfo
```

用一些更强的功能与 SQL 组合，就可以计算出更多的信息查询。

类似这些查询可以由运行在每个主机上的代理不断地计算出来。显然，这只有在把区域信息扩散给 Astrolable 的所有结点后才可能。最后，运行在主机上的代理负责计算相关区域的部分表。

这种信息交换的结果是，最后，所有参与获得聚集信息的代理都可以看到相同的结果（假设在其间没有发生变化）。

2.4.3 示例：Globule 中的差分复制策略

让我们来看看 Globule，它是一个协作的上下文分布式网络（Pierre 和 Van Steen 2006）。Globule 依赖于位于因特网中的终端用户服务器，以及那些通过 Web 页面复制来协作实现优化性能的服务器。在终端，每台初始服务器（即负责处理特定 Web 站点更新的服务器）跟踪每个页面的访问模式。该访问模式表现为一个页面的读写操作，每个操作由该页面的初始服务器添加上时间戳和日志。

在最简单的形式中，Globule 假设因特网可以看作是一个前面已经介绍过的边界服务器系统。特别是，它假定请求总是可以通过适当的边界服务器来传输，如图 2.18 所示。如果在特定的边界服务器中放置了副本，那么这种简单的模型使得初始服务器可以看到所发生的事情。把副本放置在离客户近的地方，可以改善响应延时，但这样做的缺点是，为了保持副本与初始页面的一致性，将增加初始服务器与边界服务器之间的通信流量。

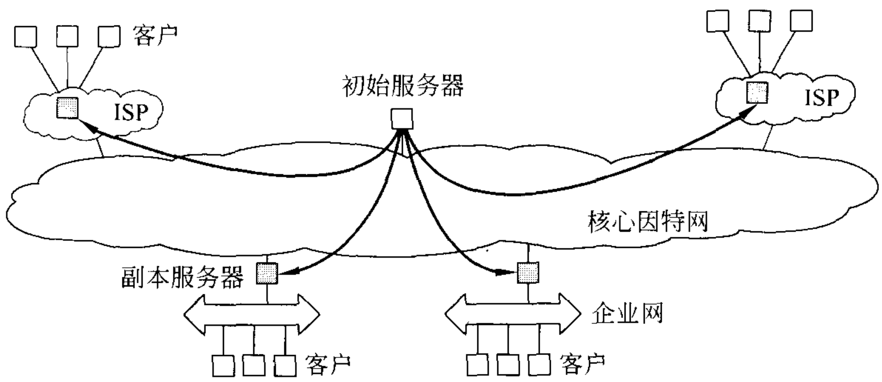


图 2.18 Globule 的边界服务器模型

当服务器接收了一个网页的请求，它会记录下该请求的来源地的 IP 地址，并使用 WHOIS 因特网协议查寻与该请求相关的 ISP 或企业网（Deutsch 等 1995）。该初始服务器然后查找可作为这个客户的边界服务器的最近的已有副本服务器，然后计算到该服务器的延时和最大带宽。在最简单的配置中，Globule 假定副本服务器与请求用户机器之间的延时是可忽略不计的，且两者之间的带宽是足够大的。

一旦收集了对一个页面的足够多请求，初始服务器就执行一个简单的“what-if 分析”。这个分析的要点是评价几种复制策略，其中每种策略描述了特定页面要复制到何处，如何保

持页面的一致性。每种复制策略发生的成本可以表示成如下的简单线性方法：

$$cost \doteq (w_1 \times m_1) + (w_2 \times m_2) + \cdots + (w_n \times m_n)$$

其中，w_k表示性能度，m_k为权重，表明这种性能度的重要程度。常见的性能度包括在客户和副本服务器之间返回 Web 页面时的总延时，初始服务器与副本服务器之间为保持复制一致性所消耗的总带宽，以及(允许)返回给客户的旧副本的数目(Pierre 等 2002)。

例如，假设在客户 C 发出请求时和页面返回给最好副本服务器之间的一般延时是 $d_C$ms。注意，这里所指的最好副本服务器是由复制策略决定的。设 m_1 是给定时间段内的总延时，即 $m_1 = \sum d_C$。如果初始服务器想要优化客户敏感的延时，它将为 w_1选择相对高的值。这样，只有那些真正能使 m_1最小化的策略才能实现相对低的成本。

在 Globule 中，初始服务器使用迹线驱动模拟方法为每个页面定期评价几十种复制策略。从这里模拟结果中选出最好的策略。这可能意味着要在不同的边界服务器中安装新的复制，或需要使用不同的方法来保持复制的一致性。迹线收集、复制策略的评价和所选策略的增强都是自动完成的。

这里还有一些需要处理的微妙问题。其中之一是，不清楚在对当前策略评价之前需要收集多少请求。解释如下：假设在 $T_i \sim T_{i+1}$时刻，初始服务器选择了策略 p。这种选择是基于 $T_{i-1} \sim T_i$时刻的请求来进行的。当然，事后在 T_{i+1}时刻，服务器认为，根据 $T_i \sim T_{i+1}$所发出的实际请求应该选择 p^*。如果 p^* 不同于 p，那么在 T_i时刻选择 p 就是错误的。

事实证明，错误的预计率与用来预测和选择下一策略的请求系列长度(称为迹线长度)有关。这种关系如图 2.19 所示。

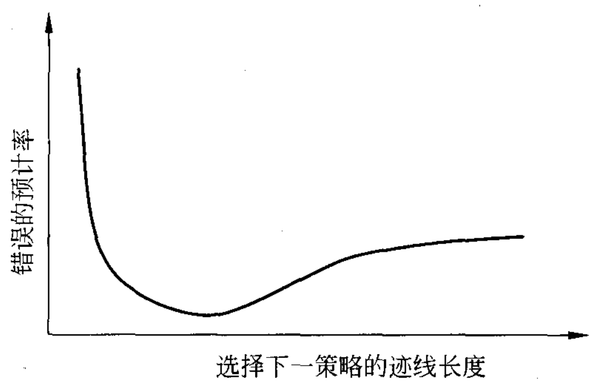


图 2.19 预计精确度与迹线长度的关系

2.4.4 示例：Jade 的自动组件修复管理

在维护每个都运行着复杂服务器的计算机群时，减轻这些计算机的管理问题变得很重要。对那些基于组件构建的服务器，可以应用的一种方法是，检测组件故障，并自动替代它们。Jade 系统就是遵循这种方法(Bouchenak 等 2005)。本节我们来介绍它。

Jade 是基于 Fractal 组件模型构建的，这是一种 Java 语言实现的框架，允许在运行时添加和删除组件(Bruneton 等 2004)。Fractal 中的组件有两种类型的接口：**服务器接口**(Server Interface)用于调用那些由组件实现的方法；**客户接口**(Client Interface)用于由一个组件调用其他组件。组件之间是用**绑定**(binding)接口相互连接起来的。例如，组件 C1 的

客户接口可以被绑定到组件C2的服务器接口。一个基本绑定的意义是，对客户接口的调用直接导致对已绑定的服务器接口的调用。在复合绑定中，该调用可能要经过一个或多个组件，因为客户和服务器接口不匹配，需要某种类型的转换。另一个原因可能是相互连接的组件位于不同的计算机中。

Jade使用了一个**修复管理域**（repair management domain）。这种域由多个结点组成，每个结点代表一个服务器以及由该服务器运行的组件。这里有一个独立的结点管理器，负责为该域添加和删除结点。为确保有高度的可用性，可以把该结点管理器备份。

每个结点都备有故障检测器，用于监测结点或其他组件的状况，并向结点管理器报告各种故障。通常，这些检测器检测的是组件的异常变化、资源使用情况，以及组件的真正故障。注意，后者可能意味着计算机已崩溃了。

当检测到一个故障时，就启动一个修复过程。这种过程由修复策略所驱动，部分由结点管理器运行。策略被明确声明，并根据已检测出的故障来运行。例如，假设已检测出了一个故障。在这种情况下，修复策略可能规定要执行以下步骤：

(1) 终止无故障结点中的组件与已发生故障结点中的组件之间的所有绑定。

(2) 请求结点管理器启动和添加新结点到该域。

(3) 配置新结点，该结点具有与已崩溃结点完全相同的组件。

(4) 重新建立前面已终止的所有绑定。

在本示例中，修复策略较简单，且只工作在关键数据没有丢失的时候（已崩溃组件称为**无状态的**（stateless））。

Jade系统使用的方法是自我管理的一个示例：根据故障检查，修复策略自动运行，把这个系统带回到崩溃前的状态。作为一个基于组件的系统，这种自动修复要求有特定的支持，以允许在运行时添加和删除组件。通常，不可能把旧的应用程序调整为自我管理系统。

本章小结

分布式系统可以以多种方式来组织。我们可以区分软件体系结构和系统体系结构。后者可以把组成分布式系统的组件放在不同的机器上。前者则更关注软件的逻辑结构：组件如何交互，以什么方式构建组件，如何实现独立性等。

关于体系结构的一个关键思想是体系结构的样式。一个样式反映了在组织构成分布式系统的软件组件之间的接口时所遵循的基本原则。重要的样式有分层、面向对象、面向事件和面向数据空间。

分布式系统的组织结构有多种。一个重要的类型是机器分成了客户和服务器。客户发送一个请求给服务器，然后，服务器生产一个结果，该结果返回给客户。客户-服务器体系结构反映了模块化软件的传统方法，其中一个模块可以调用另一个模块中的函数。通过在不同的机器上放置不同的组件，我们就可以获得多台机器上的函数的物理分布性。

客户-服务器体系结构经常是高度集中化的。在非集中化的体系结构中，我们可以看到，组成分布式系统的进程起着相同的作用，又称为点对点系统。在点对点系统中，进程组织为一个覆盖网络，这是一个逻辑网络，其中的每个进程都有一个与之通信的其他对等体列表。在覆盖网络中，可以为进程间的消息路由部署确定方案。在非结构化网络中，对等体列表多少有点是随意的，这意味着要部署查找算法来定位数据和其他进程。

另外，还开发了自我管理分布式系统。一定程度上说，这些系统融合了来自系统和软件体系结构的思想。自我管理系统通常可以组织成为反馈控制循环。这种循环含有一个监视组件、一个分析组件和多种工具集。反馈控制循环可以集成到分布式系统的很多地方。但如何开发和部署循环还需要更多的研究。

习 题

1. 如果客户和服务器相隔很远，我们可以看到，网络延时将主导整个性能。我们如何处理这个问题?
2. 什么是三层客户-服务器体系结构?
3. 垂直分布性和水平分布性的区别是什么?
4. 假设进程链 P_1，P_2，…，P_n实现了一个多层客户-服务器体系结构。进程 P_i是进程 P_{i+1}的客户，只有在从 P_{i+1}接收了一个回复后，P_i才会返回一个回复给 P_{i-1}。当我们查看进程 P_i处的请求-回复性能时，会发现这种体系结构的主要问题是什么?
5. 在结构化的覆盖网络中，消息是根据覆盖的拓扑结构来路由的。这种方法的主要缺点是什么?
6. 请看图 3.18 中的 CAN 网络。如何把坐标为(0.2，0.3)的结点的消息路由到坐标为(0.9，0.6)的结点?
7. 假设 CAN 中的一个结点知道其紧邻结点的坐标，一个合理的路由策略是把消息路由给离目的地最近的结点。这种策略的好处是什么?
8. 在非结构化的覆盖网络中，每个结点随机地选择 c 个邻结点。如果 P 和 Q 都是 R 的邻结点，那么 P 和 Q 互为邻结点的概率是多少?
9. 再来看看在非结构化的覆盖网络中，每个结点随机地选择 c 个邻结点。要查找一个文件，结点将泛洪一个请求给它的邻结点，这些请求又将再次泛洪。该请求将到达多少个结点?
10. 在点对点网络中，并不是每个结点都能成为超级对等体。满足超级对等体的合理要求是什么?
11. 在 BitTorrent 系统中，每个结点具有一个带宽为 Bout 的输出链接，以及一个带宽为 Bin 的输入连接。这些结点(称为种子结点)中的一些可以自愿地为其他结点提供下载文件。如果假设在某个时候，一个 BitTorrent 系统最多只有一个种子结点，那么它的最大下载带宽是多少?
12. 请从技术的角度来解释为什么 BitTorrent 中的 tit-for-tat 策略比因特网中的文件共享要好得多?
13. 我们给出了在自适应中间件中使用中断器的示例。请问还有些其他什么示例?
14. 中断器与其部署所在的中间件的依赖程度如何?
15. 现代的汽车都装备有电子设备。请给出在汽车中的一些反馈控制系统的示例。
16. 请给出自我管理系统的一个示例，其中的分析组件为完全分布式或隐藏的。
17. 请描述一个解决方法，为 Globule 中的预测复制自动确定最佳跟踪路径长度。
18. (**实验**)使用现有的软件，实现一个基于 BitTorrent 的系统，从单个功能强大的服务器发布文件给多个客户。使用标准的 Web 服务器可以使事情得到简化。

第3章

进　程

在本章中我们将详细讨论各种进程在分布式系统中是如何发挥重要作用的。进程的概念源自于操作系统，它定义为执行中的程序。从操作系统的角度来说，进程管理和调度也许是它要处理的最重要的问题。但是对于分布式系统来说，还存在许多同等甚至更加重要的问题。

例如，为了有效地组织客户-服务器系统，通常使用多线程技术更为方便。在分布式系统中，线程的主要作用就是以适当的方式来构建客户和服务器，使得通信和本地处理过程可以并行进行，从而获得性能上的提高。

虚拟化(virtulization)是近几年来开始流行的一个概念。它允许一个应用程序(可能连同操作系统在内的所有运行环境)并行地与其他程序一同运行，但高度独立于底层硬件和**平台**(platform)，从而达到高度**可移植性**(portability)。另外，虚拟化也有利于隔离由错误和安全问题引起的系统**失效**(failure)。我们将专用一节来特别讨论这个重要的概念。

我们在第2章中曾经指出，客户-服务器组织结构在分布式系统中是很重要的。在本章中，我们将对客户和服务器的典型组织结构作详细的剖析。另外，我们还将考察常见的服务器设计问题。

进程在不同机器之间的迁移是一个重要问题，对于广域分布式系统来说，这个问题显得特别重要。进程的迁移(更明确地说是代码的迁移)有助于获得可扩展性，也可以帮助动态地配置客户和服务器。本章将讨论代码迁移的确切含义及其相关内容。

3.1 线　　程

虽然**进程**(process)构成了分布式系统中的基本组成单元，但是实践表明，操作系统提供的用于构建分布式系统的进程在粒度上还是太大了。而就粒度而言，将每个进程细分为若干控制**线程**(thread)的形式则更加合适，可以使构建分布式应用程序变得更加方便，并可获得更好的性能。在本节中，我们将详细分析线程在分布式系统中所扮演的角色，并且说明线程的重要性。关于线程以及使用线程构建应用程序的更详细内容请参见(Lewis 和 Berg 1998)以及(Stevens 1999)。

3.1.1 线程简介

为了理解线程在分布式系统中所扮演的角色，重要的是要了解进程是什么，以及进程与线程之间的关系。为了程序执行的需要，操作系统创建多个虚拟处理器，每个虚拟处理器运行一个程序。为了保持对这些虚拟处理器的跟踪，操作系统中有一张**进程表**（process table），其包含的条目中存储着CPU寄存器值、内存映像、打开的文件、统计信息、特权信息等。进程一般定义为执行中的程序，也就是当前在操作系统的某个虚拟处理器上运行的一个程序。操作系统特别注意确保独立的进程不会有意或者无意地破坏其他独立进程运行的正确性。也就是说，多个进程并发地共享同一个CPU以及其他硬件资源这样一个事实是透明的。一般说来，操作系统需要硬件支持来实现这种隔离。

要得到这种并发透明性需要付出相对较高的代价。例如，每次创建一个进程的时候，操作系统必须分配一个完整的独立地址空间。空间分配意味着要对内存段进行初始化，比如先对数据段清零，将有关的程序复制到文本段中，随后为临时数据建立堆栈等。在两个进程之间切换CPU的开销同样会比较大，除了要保存CPU环境（包括寄存器值、程序计数器、堆栈指针等）以外，操作系统还必须修改**内存管理单元**（memory management unit，MMU）的寄存器，并且将位于**转换后备缓冲器**（translation lookaside buffer，TLB）中的地址转换缓存内容标记为无效。另外，如果操作系统支持同时运行的进程数目超出主存容纳能力，则必须在切换进程之前先在主存和磁盘之间进行交换。

类似于进程的概念，一个线程独立地执行它自己的程序代码。然而，线程与进程不同的是，如果要取得高度的并发透明性会导致性能降低的话，那么线程就不会有这样的企图。因此，线程系统一般只维护用来让多个线程共享CPU所必需的最少量信息。特别是，**线程上下文**（thread context）中一般只包含CPU上下文以及某些其他的线程管理信息。比如说，线程系统可能会记录线程当前正在某个互斥变量上阻塞的事实，这样，系统就不会选择该线程执行。那些对于多线程管理不是完全必要的信息通常被忽略。由于这个原因，在单个进程中防止数据遭到某些线程不合法的访问的任务完全落在了应用程序开发人员的肩上。

这种方法有两个重要含义。首先，多线程应用程序的性能至少不会比其单线程版本差。事实上，在很多情况下，多线程能够提高性能。其次，由于线程并不像进程那样彼此隔离并受到系统自动提供的保护的，因此多线程应用程序的开发需要付出更多的努力。设计合理并实行简单的原则会大有益处。不幸的是，现在的实践却证明这条原则并没有得到很好的理解。

3.1.1.1 非分布式系统中的线程用法

在讨论线程在分布式系统中的作用之前，首先让我们来考察它们的传统用法——在非分布式系统中的用法。在进程中使用多线程有许多好处。这就使得多线程系统的使用变得流行起来。

多线程最显著的好处来自以下事实，那就是在只拥有单线程的进程中，一旦执行了造成阻塞的系统调用，整个进程就被阻塞了。为了说明这一点，我们来考虑一下诸如电子表格这样的应用程序，并且假定用户需要连续并且交互地改变值。电子表格程序的重要特点是，需

要维护不同单元格之间的函数依赖关系，而这些单元格常常位于不同的数据表中。因此，一旦修改了某个单元格，所有关联的单元格都要自动更新。如果用户对某个单元格中的值进行改动，将会触发大量计算操作。如果只有单个控制线程，在程序等待输入的时候就无法进行计算，同时也很难在进行计算的时候接受输入。最方便的解决办法是使用至少两个控制线程：其中一个管理与用户之间的交互，而另一个进行数据表的更新工作。当这两个线程在并行工作的时候，还可以用第三个线程来备份表格数据到硬盘。

多线程的另一个优点是，在多处理器系统上执行多线程程序的时候，可以使用并行操作技术。在这种情况下，每一个线程占用一个CPU，而共享的数据则存储在共享主存中。如果设计得当，这种并行操作可以是透明的：进程可以与运行在单处理器系统上一样好，只是慢一些。随着相对便宜的多处理器工作站的出现，并行操作多线程技术变得愈发重要。这种计算机系统的典型应用是在客户-服务器应用程序中用来运行服务器程序。

多线程技术在大型应用程序上下文中也是很有用的。这种应用程序一般是作为一组相互协作的程序开发出来的，其中每一个程序都通过独立的进程执行。这种方法的典型例子是UNIX系统。程序间协作是通过进程间通信（IPC）机制实现的。对于UNIX系统来说，这套机制中通常包括已命名管道、消息队列以及共享内存段，请见（Stevens 1992）。所有IPC机制都有一个主要的缺陷，就是其中的通信需要开销庞大的上下文切换，如图3.1中三个点所示。

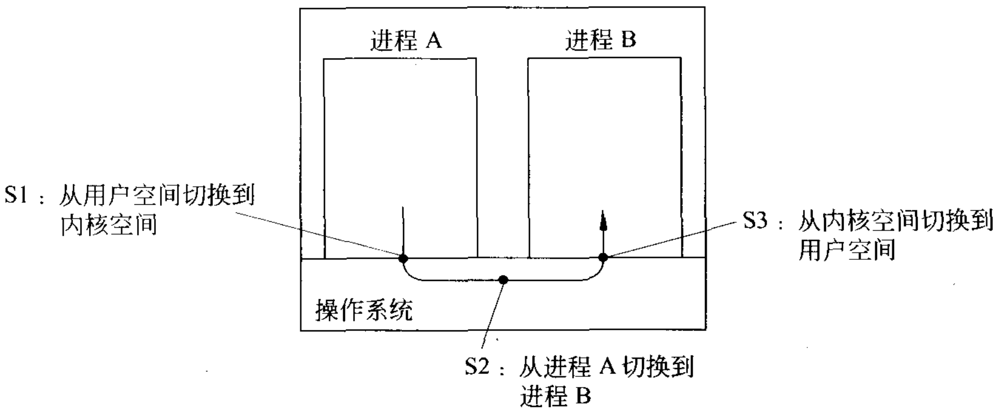


图3.1 IPC造成的上下文切换

由于IPC需要内核干预才能进行，因此，要进行IPC的进程一般首先要从用户模式切换到内核模式，如图3.1中的S1所示。这就需要改变MMU中的内存映像，同时还要刷新TLB。在内核中进行进程上下文的切换（图3.1中的S2），随后就可以从内核模式切换回用户模式以使得通信的另一方能够激活。后一次切换也同样需要改变MMU映像并且刷新TLB。

除了像上面那样使用进程来构建应用程序，我们也可以使用独立的线程来执行不同的组成部分。各个组成部分之间的通信完全通过数据共享来处理。线程切换可以完全在用户空间中完成，也可以由内核来掌管线程并对它们进行调度。这样在性能上就可以得到极大的提升。

最后，使用线程还出于纯粹的软件工程上的考虑。有许多应用程序如果用一组相互协作的线程来构建，常常是比较简便的。想象一下那些需要完成若干任务（这些任务或多或少

是独立的)的应用程序。比如说,在字处理器中,可以用独立的线程来分别处理用户输入、拼写检查及语法检查、文档布局、索引生成等工作。

3.1.1.2 线程的实现

线程一般是以线程包的形式提供的。这种包中含有创建与销毁线程的操作,以及对诸如互斥变量以及条件变量之类的同步化变量的操作。有两种实现线程包的基本方法:第一个方法是可以构造一个完全在用户模式下执行的线程库;另一个方法是由内核来掌管线程并进行调度。

采用用户级线程库有很多好处。首先,创建和销毁线程的开销很小。由于所有线程管理工作都保持在用户地址空间中进行,线程创建的开销主要取决于为线程堆栈的建立而分配内存的开销。与此类似,销毁线程的工作主要是释放线程堆栈所占用的内存,因为线程销毁后它将不会再使用这些内存。这些操作的开销都不大。

采用用户级线程的好处还有,可以通过为数不多的几条指令来实现线程上下文的切换。基本上,只有 CPU 寄存器值需要存储,并随后用将要切换到的线程的原先存储的值重新加载到 CPU 寄存器中去,并不需要内存映像的改变、TLB 的刷新以及 CPU 统计等。线程上下文的切换是在两个线程需要同步的时候发生的,比如说在进入共享数据段的时候。

然而,用户级线程的主要缺陷在于,对引起阻塞的系统调用的调用将会立即阻塞该线程所属的整个进程,也就阻塞了所属进程中的所有其他线程。我们曾经说过,如果要将大型应用程序分为若干部分来构建,所有部分在逻辑上可以同时执行,那么线程是非常有用的。在这种情况下,在 I/O 过程中发生的阻塞不应该妨碍此时其他部分的执行。对于有这种需要的应用程序,用户级线程没什么益处。

这些问题大都可以通过在操作系统的内核实现线程的方法来解决。不幸的是,这种解决方法的代价非常大:每一个线程操作(创建、删除、同步化等)都必须由内核来执行,需要进行系统调用。线程上下文的切换的开销可能变得与进程上下文切换的开销一样大。结果,用线程代替进程的大多数优点都不复存在了。

一种解决方法是采用用户级线程和内核级线程的混合形式,一般称为**轻量级进程**(lightweight processes,LWP)。LWP 运行在单个重量级进程的上下文中,每个进程中可以包含多个 LWP。除了 LWP 以外,系统还提供用户级线程包,向应用程序提供了创建和销毁线程等普通操作。另外,包中还提供了用于线程同步的工具,比如互斥变量和条件变量。重要的是,线程包是完全在用户空间中实现的。也就是说,执行这些线程操作不需要内核的干预。

线程包可以由多个 LWP 共用,如图 3.2 所示。这就意味着每个 LWP 可以运行自己的用户级线程。建立多线程应用程序时首先要创建用户级线程,随后分配每个用户级线程到一个 LWP。将线程分配给 LWP 的行为一般是隐式的,并且向程序员隐藏。

用户级线程和 LWP 一般是通过以下方式结合起来:线程包中有一个用于调度线程的例程。在创建 LWP(一般通过系统调用来创建)的时候,LWP 得到了自己的堆栈,并且执行调度例程,该调度例程会寻找下一个线程来执行。如果有多个 LWP 存在,每一个 LWP 都会执行该调度例程。用来跟踪当前线程集的线程表是由各 LWP 共享的。通过完全在用户空间中实现的互斥标志来对该表进行保护,以确保只能对它执行互斥访问。也就是说,各

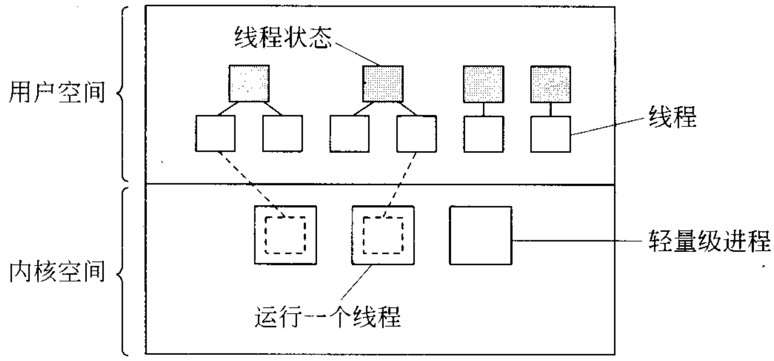


图 3.2 将内核模式的轻量级进程与用户级线程结合使用

LWP 间的同步并不需要任何内核支持。

如果 LWP 找到了一个可运行的线程，它就将上下文切换到该线程。同时其他 LWP 也会寻找其他可运行的线程。如果线程由于互斥变量或者条件变量需要阻塞，它就会在完成必要的管理工作之后调用调度例程。如果找到了另一个可运行的线程，就会将上下文切换成该线程拥有的上下文。这种方法的优点在于，对执行用户级线程的 LWP 来说，用户级线程上下文切换只是普通的程序代码，因为上下文切换完全在用户空间中实现。

现在，我们来看一下线程进行阻塞系统的调用时的情形。在这种情况下，执行过程不再处于用户模式，而是转变到内核模式中，但是仍然在当前的 LWP 上下文中继续执行。在当前的 LWP 无法继续执行的时候，操作系统会将上下文切换到另一个 LWP，这就意味着上下文会重新切换回用户模式。被选中的 LWP 将会简单地从上一次停止的地方继续执行。

将 LWP 与用户级线程包结合起来使用有很多优点。首先，线程创建、销毁及同步化工作的开销相对较小，并且根本不需要内核干预。其次，如果某个进程中有足够数量的 LWP，则阻塞的系统调用将不会导致整个进程被挂起。第三，应用程序并不需要知道 LWP 的存在，它能见到的只是用户级线程。最后，通过在不同 CPU 上执行不同 LWP，LWP 可以在多处理器系统中方便地应用。拥有多个处理器这个事实可以对应用程序完全隐藏。轻量级进程与用户级线程组合的唯一缺点是，仍然必须进行 LWP 的创建和销毁工作，这个工作的开销并不比内核级线程的小。然而值得庆幸的是，只是偶尔需要进行 LWP 的创建和销毁工作，而且受到操作系统的完全控制。

另一种与轻量级进程相似的方法是使用**调度例程激活**（scheduler activation）（Anderson 等 1991）。调度例程激活和 LWP 之间的本质区别在于，在调度例程激活中，当线程在系统调用中被阻塞时，内核对线程包进行**上行调用**（upcall），调用调度例程来选择下一个可执行的线程来执行。当线程释放的时候，也要重复相同的步骤。然而，使用上行调用是一种不太好的做法，因为它破坏了分层系统的结构，在分层系统中只允许对紧接着的下一层进行调用。

3.1.2 分布式系统中的线程

线程的重要特性之一是，它提供了一种方便的方式允许使用会导致阻塞的系统调用而不阻塞该线程所属的整个进程。这种特性使得在分布式系统中使用线程变得特别有吸引

力，因为利用它可以极为方便地将通信表述为同时维护多个逻辑连接的形式。我们将通过对多线程的客户和服务器分别进行详细考察来说明这一点。

3.1.2.1 多线程客户

为了确立高度的分布透明性，在广域网上构建的分布式系统需要隐藏较长的进程间消息传播的时间。在广域网中，传输的延迟很容易达到上百毫秒，甚至几秒。

隐藏通信时间延迟的常规方法是启动通信后立即进行其他工作。这方面典型的例子是Web浏览器。在很多情况下，Web文档是由HTML文件组成的，HTML文件中包含有纯文本文件以及图像组、图标等。为了获得Web文档中每一个组成部分，浏览器必须建立TCP/IP连接，读取输入数据并将数据传递给显示组件。建立连接和读取输入的数据在本质上都是可能导致阻塞的操作。在进行远程通信的时候，还要考虑每个操作所花费的时间可能会相对很长。

Web浏览器一般在开始获取HTML页面后随即就显示它。为了尽量隐藏通信延时，某些浏览器在接收数据的过程中就开始显示这些数据。首先将文本显示出来以便用户查看，并且提供页面滚动之类的功能，同时继续获取组成页面的其他文件，比如图像等。在收到这些文件之后再显示它们。用户不必等待浏览器取得整个页面的所有组件就能够查看页面。

从效果上来说，看起来就好像Web浏览器在同时进行多项任务一样。实践已经证明，以多线程客户的模式来开发浏览器可以显著地使问题得到简化。只要取得了主HTML文件，就可以激活多个独立的线程，它们分别负责取得页面的各个部分。每个线程都与服务器建立一个独立连接以获取数据。只要假定进行导致阻塞的调用不会将整个进程挂起，与服务器建立连接和读取数据的过程就可以使用标准的(可能导致阻塞的)系统调用来编制。(Stevens 1998)中也说明了，用于每一个线程的代码都是相同的，并且首先是简单的。在浏览页面期间，用户只会注意到图像等内容显示上的延迟，但是可以对文档进行浏览。

在可以同时打开多个连接的情况下，使用多线程的Web浏览器还有另一个明显的好处。在上面的例子中，建立了到同一个服务器的多个连接。但是，如果该服务器的负载过重，或者比较慢，这种方法与逐个获取组成页面的文件的方法相比，在性能上并不会有什么真正的提升。

然而，在许多情况下，Web服务器会复制到多台机器上，每个服务器负责提供一组完全相同的Web文档。复制的服务器位于同一个站点，并且名字是相同的。当对Web页的访问请求到来时，该请求被转发到其中的一个服务器，转发到哪一个服务器的决策通常是通过循环策略或者某些其他负载平衡技术做出的(Katz等 1994)。在使用多线程客户的时候，可以与不同的服务器副本建立连接，这样就可以并行地进行数据传输了，并且确保整个Web文档完全显示出来所需的时间与使用无复制的服务器的情况相比要短得多。这种方案只在客户真正能够处理输入数据的并行流时才能够发挥作用，而使用线程正是达到这个目的的理想方式。

3.1.2.2 多线程服务器

虽然我们已经看到多线程客户有许多优点，但是分布式系统中多线程技术主要还是在

服务器端发挥作用。实践表明，多线程技术不仅能够显著简化服务器代码，还能够使得应用并行技术来开发高性能的服务器变得更加容易，即使在单处理器系统上也是如此。而且，目前多处理器计算机广泛地用作通用工作站，所以多线程技术就更加有用了。

为了理解使用线程给编写服务器代码带来的好处，考虑一下文件服务器的组织结构，该文件服务器可能会偶尔由于等待磁盘操作而阻塞。文件服务器一般等待输入的文件操作请求，随后执行该请求，最后送回应答。其中一种特别流行的组织结构如图3.3所示。图中有一个称为**分发器**(dispatcher)的线程，由它来读取文件操作请求。客户发送请求到服务器的某个已知端点。在对请求进行检查以后，服务器选择一个空闲的（也就是阻塞的）**工作者线程**(worker thread)，由它来处理该请求。

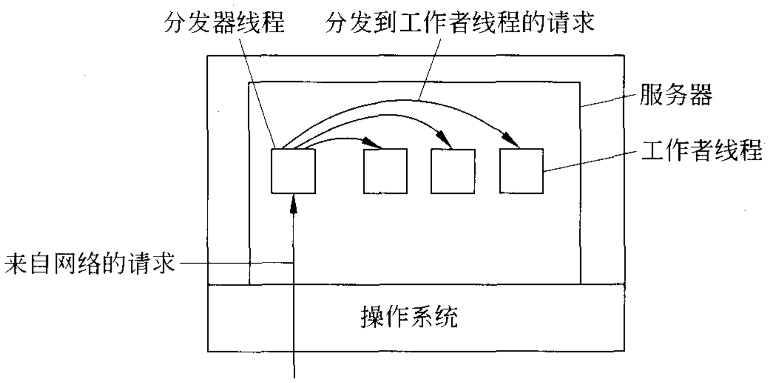


图3.3 以分发器/工作者模型组织起来的多线程服务器

工作者线程在本地文件系统上执行阻塞的read调用，执行该调用将会导致该线程被挂起直到数据从磁盘上读出为止。如果该线程被挂起了，就选择另一个线程接着执行。比如说，可以选择执行分发器线程以完成更多工作。此外，也可能选择另一个准备就绪的工作者线程来运行。

现在来考虑在不使用多线程的情况下文件服务器的编写方式。一种可能是让服务器作为单个线程来工作。由文件服务器主循环程序来获得请求，检验该请求，随后执行该请求直至完成，然后再接受另一个请求。在等待磁盘操作的时候，服务器是空闲的，但也不对其他请求进行处理。也就是说，来自其他客户的请求得不到处理。另外，如果文件服务器运行于专用机器上（大多数情况下是如此），文件服务器等待磁盘操作的时候CPU将会完全空闲。这样做的结果是，服务器每秒能够处理的请求数目大大减少。而使用多线程能够显著提升性能，而且每个线程都是以通常的方式编写成顺序执行的。

至此我们已经看到了两种可行的设计方法：多线程文件服务器以及单线程文件服务器。假定无法使用多线程，但是系统设计者又发现使用单线程所带来的性能损失是不可接受的，那么第三种可行方法是将服务器作为一个大的有限状态机来运行。当请求输入的时候，由唯一的线程对请求进行检查，如果可以利用缓存中的内容来满足请求当然好，如果缓存的内容无法满足就必须向磁盘发出消息，请求执行磁盘操作。

然而，向磁盘发出消息之后线程并不阻塞，而是在状态表中记录下该请求的当前状态，随后接收下一个消息。下一个消息可能是一个开始新工作的请求，也可能会是磁盘对之前的操作请求所做的应答。如果是新工作请求，就进行该工作。如果是来自磁盘的应答，就从表中取出之前记录下的那个请求的有关信息，对应答进行处理并把结果发送给客户。如果

采用这种方法，服务器必须使用非阻塞性的 send 和 receive 调用。

在这种设计中，前两种方法使用的"顺序处理"模型不见了。对于每一个发送和接收的消息来说，计算状态都必须显式地保存下来，存储在表中。从效果的角度来看，这么做是在以一种复杂的方法模拟线程及其堆栈。进程是作为有限状态机来运行的，它在接收到一个事件之后根据其内容来决定采取的操作。

现在，多线程的作用已经很清楚了。多线程能够保留顺序处理的思路，使用阻塞性的系统调用（比如，RPC 与磁盘通话），并且仍然能达到并行处理的目的。使用阻塞性系统调用能使编程更加容易，而并行处理能够提升性能。单线程服务器保留了使用阻塞性系统调用所带来的方便之处，但是放弃了性能。而有限状态机的方法能够通过并行获得高性能，但是由于使用非阻塞性系统调用，编程上比较困难。这些模型总结在图 3.4 中。

模型	特征
多线程	并行，使用会导致阻塞的系统调用
单线程进程	非并行，使用会导致阻塞的系统调用
有限状态机	并行，使用非阻塞系统调用

图 3.4 构建服务器的 3 种方式

3.2 虚 拟 化

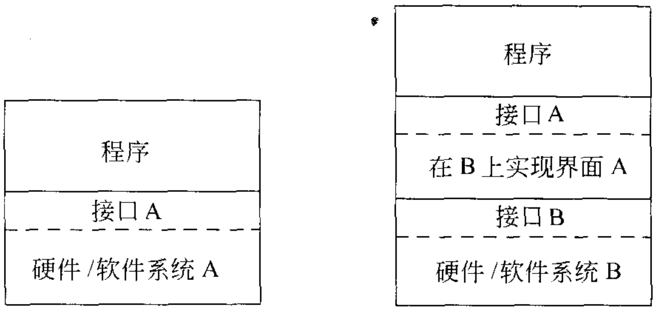
线程和进程提供了一种同时做多件事情的方式。从效果上来说，它们可以用来构建看起来像同时执行的程序。在单处理器的计算机上，这种同时执行当然只是一种感觉。由于只有一个处理器，任一时间只能是某个线程或进程一条指令在执行。通过在线程和进程之间快速切换，给人的感觉是多个线程和进程在同时执行。

这种只有单个处理器但感觉有多个处理器机制可以扩展到其他资源，导致所谓的**资源虚拟化**(resource virtualization)。这种概念在过去几十年里也应用过。但只是当分布式计算机系统更广泛的使用并且更复杂后，虚拟化才重新变得热起来。这也表明应用软件通常比底层的系统软件和硬件更有生命力。这一节中我们考察虚拟化的作用并且讨论它的实现。

3.2.1 虚拟化在分布式系统中的作用

如图 3.5(a)所示，基本上每个分布式计算机系统都提供一个给上层软件的界面。界面可以有多种，从处理器提供的基本指令集到很多中间件系统提供的巨大的应用程序界面集。如图 3.5(a)所示，虚拟化本质上是扩展或替换一个现存界面来模仿另一个系统的行为。我们将会讨论虚拟化的技术细节。但首先考察一下为什么虚拟化对分布式系统很重要。

在 20 世纪 70 年代提出虚拟化的最重要原因是让老的软件可以在新的昂贵的大型机上运行。那些老的软件不仅包括应用软件，也包括了运行这些软件的操作系统。这种支持老软件方法在 IBM370 和后继机上得到成功的应用。IBM370 和后继机提供一个虚拟机并把多个不同的操作系统移植到它上面。



(a) (b)

图 3.5
(a) 程序、接口与系统之间的组织结构;(b) 系统A位于系统B之上的组织结构

随着硬件变得便宜,计算机功能更强大,并且操作系统种类变少,虚拟化也变得不那么重要。然而,从20世纪90年代期以来,有几个原因促使事情又一次发生了变化。

第一,尽管硬件和低层系统软件变化得比较快,高层软件(比如中间层和应用软件)却稳定得多。也就是说,我们面对的一种情况是旧有软件(legacy software)的维护跟不上下层平台更新的步伐。通过移植旧有软件的底层接口到新平台,虚拟化可以帮助解决这个问题。另外,这也使得一大类的现有软件可以在立刻新平台上用。

另一个同等重要的事实是网络已经无所不在。很难想象有一台没有联网的现代计算机。在实践中,这种无所不在的网络互联使得系统管理员要维护很多不同的服务器。每台服务器运行不同的应用程序,可以被客户访问。同时多种资源应该很容易地被应用程序访问。虚拟化可以提供很大帮助:通过让每个应用程序运行在自己的虚拟机上(可能包括运行在一个通用平台上的相关的库和操作系统),平台和机器的种类可以减少。

后一种虚拟化提供了高度的移植性和灵活性。例如,为实现能轻易支持动态内容(content)复制的内容传输(content delivery)网络,(Awadallah 和 Rosenblum 2002)论证,如果边界服务器(edge server)能支持虚拟化,允许动态复制一个站点(site)及其环境,管理将会容易得多。如我们将要讨论的一样,主要是这些移植性的好处使得虚拟化成为分布式系统的重要机制。

3.2.2 虚拟机体系结构

在实践中有多种实现虚拟化的方法。(Smith 和 Nair 2005)简要描述了这些方法。为理解不同的虚拟化技术,首先要认识到计算机系统通常在4个不同层次提供4个不同界面:

(1) 由**机器指令**(machine instruction)组成,可由任何程序激起的硬件软件界面。

(2) 由机器指令组成,只有特权程序(像操作系统)才可激活的硬件软件界面。

(3) 由操作系统提供的**系统调用**(system call)组成的界面。

(4) 由库调用组成的界面,通常形成了所谓的**应用程序编程接口**(application programming interface, API)。很多情况下前述的系统调用由API隐藏。

图3.6显示了这些不同的界面。虚拟化的实质是模仿这些界面的行为。

虚拟化可采用两种方式。第一,可以构建一个运行时(runtime)系统,实质上提供一套抽象指令集来执行程序。指令可以被翻译执行(像Java运行环境),也可以仿真执行,就像

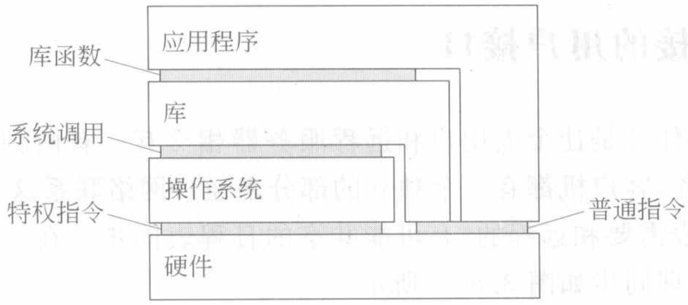


图 3.6 计算机系统多种界面

在 Unix 平台上运行 Windows 应用程序。在后一种情况，仿真器必须模仿系统调用的行为，而这已被证明远非寻常。这种虚拟化被 Smith 和 Nair 称为**进程虚拟机**（process virtual machine），用以强调虚拟化实质上作用在单个进程上。

另一种虚拟化方式是提供一种系统。把它做成一层完全屏蔽硬件但提供一个同样指令集（或其他硬件）的界面。关键是这个界面可以同时提供给不同的程序。结果，可以有多个不同的操作系统独立并发地运行在同一平台。这个层通常叫**虚拟机监视器**（virtual machine monitor，VMM）。这种方法的典型例子是 VMware（Sugerman 等，2001）和 Xen（Barham 等，2003）。这两种不同方法显示在图 3.7 中。

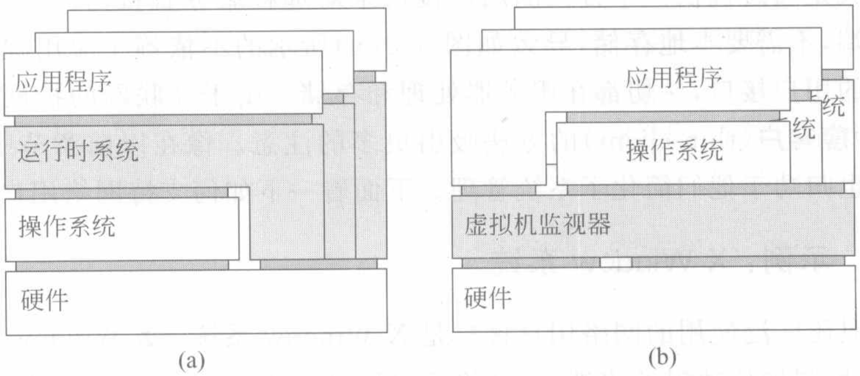


图 3.7

(a) 进程虚拟机，混合多个应用程序和运行时系统；(b) 虚拟机监视器，混合多个应用程序和操作系统

像 Rosenblum 和 Garfinkel 争论那样，虚拟机监视器将会变得对分布式系统的可靠性和安全越来越重要。由于它们能分离一个应用程序和它的环境，由错误或安全攻击引起的失败不再会影响整台机器。另外，像我们以前提过，由于虚拟机监视器进一步分离了硬件和软件，允许一个完整环境从一台机器移植到另一机器，移植性得到很大改善。

3.3 客 户

在前面的章节中我们讨论过客户-服务器模型、客户和服务器所扮演的角色以及它们之间的交互。让我们对客户和服务器分别进行详细剖析，在本节中我们先讨论客户，服务器将在下一节中讨论。

3.3.1 网络连接的用户接口

客户机器的主要任务是让个人用户和远程服务器相交互。有两种方式支持这种交互。第一，对每种远程服务，客户机都有一个独立的部分来通过网络联系这种服务。典型例子是用户 PDA 上的日程表需要和远程的、有可能共享的日程表同步。在这种情况下，一个应用级别的协议就可以处理同步如图 3.8(a)所示。

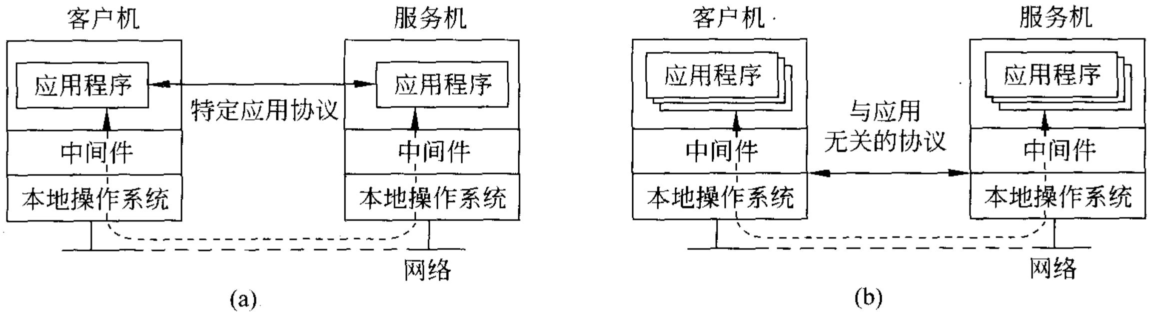


图 3.8

(a) 带自用协议的网络连接的应用程序；(b) 允许访问远程应用程序的通用解决方案

第二种方式是通过提供一个方便的用户接口来对远程服务直接访问。从效果上，客户机只是作为终端，不需要本地存储，导致如图 3.8(b)所示的不依赖于应用程序的方案。如果是网络连接的用户接口，一切都在服务器处理和存储。由于互联网的扩展和手持设备变得更成熟，这种**瘦客户**(thin-client)的方法吸引更多的注意。像在以前章节中讨论的，瘦客户方案的流行也归功于他们简化了系统管理。下面看一下如何支持网络用户接口。

3.3.1.1 示例：X Window 系统

一个很老但还广泛使用的网络用户接口是 X Window 系统。X Window 系统通常也简称为 X，它用来控制按位映射的终端，这种终端包括显示器、键盘以及鼠标之类的定点设备。从这种意义上说，X 可以看作操作系统中负责控制终端的一部分。系统的核心由称为 X 内核(X kernel)的部分构成，它包括该类终端专用的全部设备驱动程序，而且对硬件有高度的依赖性。

X 内核提供相对低级的屏幕控制接口，也负责捕获键盘和鼠标事件。接口放在称为 Xlib 的库中供应用程序调用，其一般的组织结构如图 3.9 所示。

X 有一个特性非常有趣：X 内核和 X 应用程序并不一定要驻留在同一台机器上。应该特别指出的是，X 提供了 X 协议，该协议是一个面向网络的通信协议，通过该协议 Xlib 可以与 X 内核交换数据和事件。比如 Xlib 可以请求 X 内核创建或杀死窗口，设置颜色，定义鼠标显示类型。相应地，对本地事件如键盘和鼠标输入，X 内核会发送事件给 Xlib 而做出响应。

多个应用程序可同时与 X 内核通信。其中一个叫窗口管理器的应用程序有特殊权力。它能决定显示器给用户的“外观和效果”(look and feel)。比如，它能规定如何用额外的按钮来修饰窗口，如何在显示器上摆放多个窗口等。其他应用程序必须遵守这些规则。

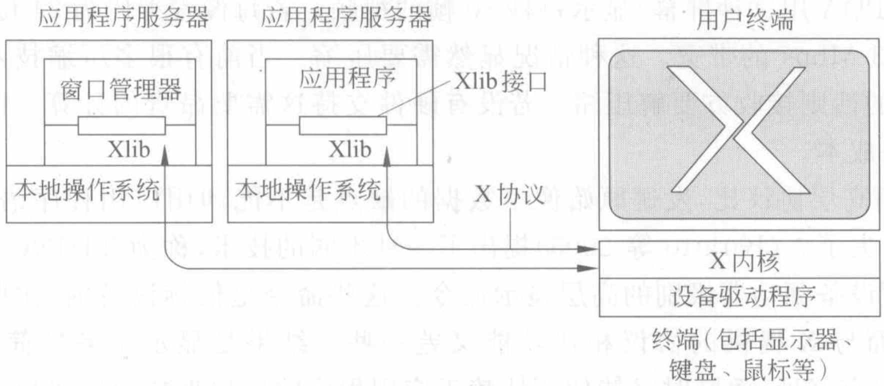


图 3.9　X Window 系统基本组织结构

下面来看看 X Window 是如何适应客户-服务器计算的。从以上描述，应该清楚 X 内核是接受应用程序(可能是远程的)请求来操纵显示器的。从这个意义说，X 内核相当于服务器，而应用程序相当于客户。X Window 已采用这些术语。尽管严格说是正确的，但也容易造成混淆。

3.3.1.2　瘦客户网络计算

应用程序使用 X Window 提供的特殊显示命令来操纵显示器。这些命令通过网络传送，随后由 X 内核执行。本质上说，X Window 上的应用程序最好把程序逻辑和用户接口命令分开。不幸地，通常不是这样做的。(Lai 和 Nieh 2002)报导，很多程序的逻辑和用户接口紧密耦合在一起的，意味着应用程序将发送很多请求给 X 内核，然后在做下一步之前期待着 X 内核的响应。当操作在有长时间延迟的广域网时，这种同步行为对性能有不利影响。

对这个问题有几种解决方案。一是像 NX(Pinzari，2003)一样改写 X 协议的实现。其中一个重要的部分是通过压缩 X 消息减少带宽要求。首先，消息由作为**标识符**(identifier)的固定部分和一个可变部分组成。很多情况下多个消息有相同标识符并且常常包含类似的数据，结果只需发送消息之间的不同部分。

发送方和接收方都维持一个本地缓存，其中的条目可由消息标识符来搜索。当发送一条消息时，首先搜索本地缓存。若有以前发送的消息具有相同标识符(数据可能不同)，使用**差分编码**(differential encoding)发送这两条消息的不同部分。接收方也对收到的消息搜索本地缓存，然后对不同部分进行解码。若搜索失败，则使用标准压缩技术，这已经能达到 4 倍带宽的减少了。总之，这种技术能达到 1000 倍的带宽减少，允许 X 在仅仅 9600 kbps 的低带宽网线上运行。

消息缓存的重要副作用是发送方和接收方共享显示器的当前状态信息。比如，应用程序可请求搜索本地缓存来获得很多不同对象的几何信息。单是共享信息就已减少了维持应用程序和显示器同步的消息。

尽管有这些改进，X 还是要求运行一个显示服务器。这可能开销过多，特别是如果显示器只是一个手机屏幕。一种解决方案让应用程序做所有处理而保持显示器的软件很简单。这意味着应用程序在像素级别上控制整个显示器。**位图**(bitmap)上的变化通过网络送给显示器并立刻传送给显示器本地的帧缓冲区。

为了不让带宽成为问题，这种方法要求有很成熟的压缩技术。比如，考虑在 320×240

的屏幕（很多PDA用这种屏幕）显示每秒30帧的视频。若每像素编码为24位，如果无压缩的话需要约53 Mbps的带宽。这种情况显然需要压缩。当前有很多压缩技术在使用。注意，若用压缩的话则接收方要解压缩。若没有硬件支持这需要昂贵的计算。而提供硬件支持会增加设备成本。

与X使用高层协议比，发送原始像素数据的缺点是不能利用应用程序语义，因为这已在位图层级丢失了。(Baratto等 2005)提出了一种不同的技术，称为THINC。他们提供几条运行在视频设备驱动器级别的高层显示命令。这些命令是依赖设备的，比原始像素操纵功能更强大，而与X提供的协议相比功能又差一些。结果是显示服务器简单得多，这对CPU利用率很有好处，而同时又能使用依赖于应用程序的优化来减少带宽和同步。

在THINC中，应用程序的显示请求被截取和翻译成底层命令。通过截取应用程序请求，THINC能利用应用程序语义翻译成最好的底层命令组合。翻译的命令并不立刻送往显示器，而是放入队列。通过批处理多个命令，有可能把它们合成单个命令，使得消息更少。比如，当一条画一个屏幕区的新命令覆盖了一条以前建立这个区但还在队列中的命令时，后者就不需要送给显示器。最后，一旦有显示器更新，THINC总是尽量送出它们，而不是等显示器来请求。

最终结果是THINC用的方法提供了更好的综合性能，尽管与NX所展示的是一致的。详细的性能比较可在(Baratto等 2005)中找到。

3.3.1.3 复合文档

现代用户接口的功能要比X这样的系统强得多。特别是，现代用户接口允许应用程序共享单个图形窗口，并在用户操作下利用该窗口来在应用程序间交换数据。用户可以进行的操作一般包括**拖放式**(drag-and-drop)操作和**在位编辑**(in-place editing)。

拖放功能的典型例子有：将代表文件A的图标移到代表垃圾箱的图标上去以删除该文件。这种情况下，用户接口所做的不仅仅是重新排列和显示图标这么简单的事情：它必须在将代表A的图标移到代表垃圾箱应用程序的图标的时候，将文件A的名字传递给与垃圾箱相关的应用程序。我们还可以想出更多此类的例子。

在位编辑可以通过包含文本和图形的文档的例子来说明。想象一下，如果该文档在标准字处理器中显示出来，只要用户把鼠标指针放到某幅图片上，用户接口就必须将有关信息传递给负责绘画的应用程序，以便用户修改该图片。比如说，用户可能想要旋转该图片，这会影响图形在文档中的布局。用户接口会计算出图片新的高度和宽度，随后将有关信息传递给字处理器，随后字处理器自动更新文档的页面布局。

隐含在这些用户接口背后的关键思想是**复合文档**(compound document)的概念。复合文档可以定义为一组文档，这些文档的类型可以不一致（比如文本、图片、数据表等），但是在用户接口层上做到了无缝集成。可以处理复合文档的用户接口隐藏了这样一个事实：文档的不同部分实际上是由不同应用程序来处理的。而对于用户来说，文档中所有部分都是无缝地集成在一起的。当某一部分的改变影响别的部分时，用户接口可采取合适的措施，比如通知相关的应用程序。

与X Window系统的情况相类似，与复合文档相关联的应用程序也不必在客户机器上执行。然而，很明显，支持复合文档的用户接口必须比不支持复合文档的用户接口做更多的

处理工作。

3.3.2 客户端软件与分布透明性

客户端软件不仅仅包含用户接口。在许多情况下,客户-服务器应用程序中的部分处理和数据级工作是在客户端执行的。嵌入式客户软件就是这样一类特殊的软件。它用于自动取款机(ATM)、现金记录器、条形码阅读机、电视机顶盒等。在这些情况下,用户接口与本地处理和通信功能相比较,在客户端软件中所占的比重很小。

除了用户接口和其他应用相关软件之外,客户端软件中还包含用于获得分布透明性的组件。理论上说,客户端不应该察觉到它与远程进程的通信。相反,对于服务器来说,出于性能和正确性方面的考虑,分布常常不那么透明。比如说,在第6章中我们将介绍,复制的服务器有时需要相互通信,以确定操作应该按照什么顺序在各服务器副本上执行。

访问透明性通常是由客户端存根处理的,该存根由服务器提供的接口定义生成。存根提供的接口与服务器所提供的相同,但是前者隐藏了不同机器的体系结构上以及实际通信上可能存在的差异。

有几种不同的方法可以用来处理定位透明性、迁移透明性以及重定位透明性。关键是要使用一个便利的命名系统,这点我们将在下一章中介绍。很多时候与客户端软件的协作也是很重要的。比如说,如果客户已经绑定到服务器,改变服务器的位置时就可以直接通知它。这时可以由客户的中间件负责对用户隐藏服务器的当前位置,并且在必要的时候透明地重新绑定到服务器上。即使在最坏的情况下,也只不过是客户应用程序会察觉到暂时的性能下降而已。

与上面的方法相类似,许多分布式系统利用客户端解决方案来实现复制透明性。比如说,想象一下一个带有副本服务器的分布式系统。可以通过将调用请求转发给每一个服务器的副本来达到复制透明性,如图3.10所示。客户代理将会透明地搜集所有对像的响应,并且只向客户应用程序送回一个返回值。

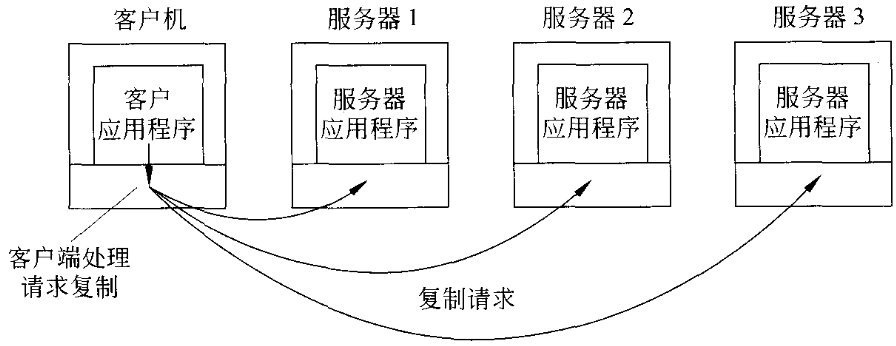


图3.10 使用客户端解决方案来透明地副本服务器

最后,考虑一下故障透明性。对服务器通信故障的屏蔽一般是通过客户中间件完成的,比如说可以对客户中间件进行配置,使它不断尝试连接到某服务器,或者在进行几次尝试(并失败)后尝试连接到其他服务器。有时甚至可以让客户中间件返回它在前一次与该服务器的会话中缓存的内容,Web浏览器在无法连接到某服务器时就是这样处理的。

并发透明性可以通过专门的中介服务器来实现，特别是可以由事务监视器来实现，而且需要客户软件的支持很少。同样，持久性透明性一般完全由服务器来处理。

3.4 服 务 器

现在我们来详细讨论服务器的组织结构。下面，我们首先重点讨论服务器上的几个常见的设计问题，随后讨论服务器集群。

3.4.1 常见的设计问题

服务器是实现特定服务的进程，这些服务是为一组客户提供的。本质上，每个服务器的组织方式都一样：等待来自客户的请求，随后负责处理该请求，最后等待下一个输入的请求。

服务器有几种不同组织结构。**迭代服务器**(iterative server)自己处理请求，并且在必要的情况下将响应返回给发出请求的客户。**并发服务器**(concurrent server)并不自己处理请求，而是将请求传递给某个独立线程或者其他进程来处理，自身立即返回并等待下一个输入的请求。多线程服务器就是并发服务器的一个特例。并发服务器还有另一种实现形式，那就是每收到一个输入请求都派生出一个新进程来对其进行处理。许多 UNIX 系统采用了这种方法。在这种方法中，由处理请求的线程或者进程负责向发出该请求的客户返回响应。

另一个问题是客户联系服务器的地方。客户总是向服务器所在机器上的端点(end point)发送请求，端点也称为**端口**(port)。每个服务器都监听一个特定的端口。那么客户如何知道某个服务所对应的端点呢？一种方法是为已知的服务分配一个统一的端点。比如说，处理因特网 FTP 请求的服务器总是在 TCP 端口 21 上监听的。同样，万维网 HTTP 服务器总是在 TCP 端口 80 上监听。这些端口是由因特网号码分配管理局(IANA)来组织分配的，存档在(Reynolds 和 Postel 1994)中。由于已分配的端口是已知的，客户只需要找出运行服务器的机器的网络地址就可以了，而网络地址可以通过命名服务来取得，我们将在下一章中说明这一点。

有许多服务不需要预先分配好端点。比如说，一个时间服务器可能会使用由本地操作系统动态分配给它的端点。在这种情况下，客户首先必须查询到该端点号。一种解决方法在运行服务器的每台机器上都运行一个特殊的守护程序，该守护程序负责跟踪位于同一台机器上的服务器实现的每一项服务所使用的当前端点。守护程序另外还监听一个已知的端口。客户通过这个端口与该守护程序进行联系，请求得到指定服务器的端点号，随后再与该服务器进行联系，如图 3.11(a)所示。

通常会把某个端点与特定的服务关联起来。然而，如果每一项服务都由单独的服务器来实现，无疑是对资源的一种浪费。比如说，在典型 UNIX 系统中，一般同时运行多个服务器，这些服务器大多被动地等待客户端的请求输入。与其对这么多被动进程进行跟踪，不如由一个**超级服务器**(superserver)来负责监听所有与这些服务关联的端点更有效率，如图 3.11(b)所示。UNIX 中的 inetd 守护程序就采用了这种方法。inetd 监听许多因特网服

务的已知端口。当收到请求的时候，它派生出一个进程以对该请求进行进一步处理，这个派生出的进程在处理完毕后将会自动退出运行。

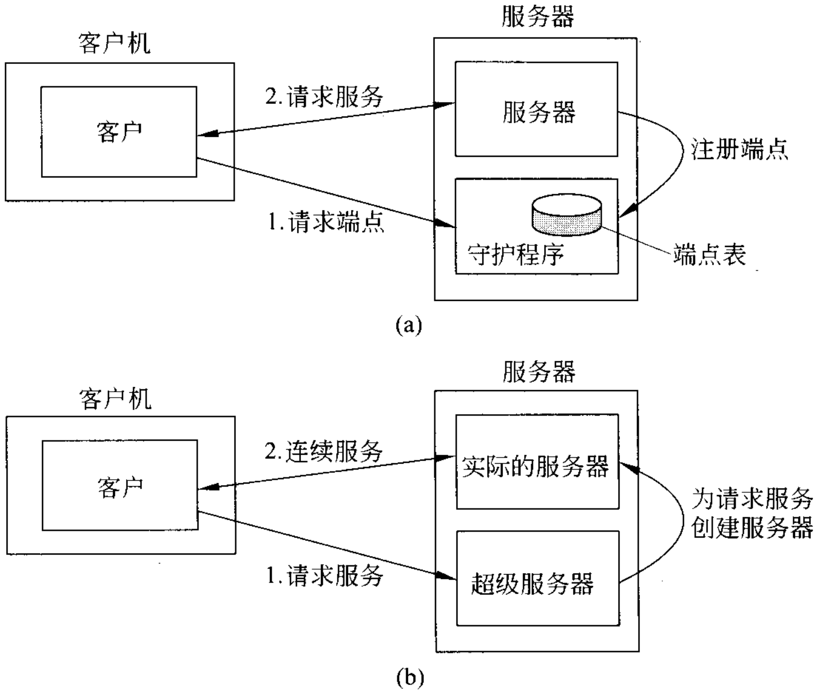


图 3.11

(a) 利用守护程序来进行客户-服务器绑定；(b) 使用超级服务器来进行客户-服务器绑定

在设计服务器时需要考虑的另一个问题是，是否可以以及怎样中断服务器的工作。比如说，考虑一下这种情况：用户刚刚决定向 FTP 服务器上载一个庞大的文件。突然该用户意识到选错了文件，想要中断服务器程序，取消数据传输。有几种方法可以做到这一点。有一种方法在当前的因特网环境下工作得非常好(有时这是唯一的可选方法)，那就是让用户立即强行退出客户应用程序(这将自动中断与服务器之间的连接)，然后马上重新启动客户应用程序，就好像什么都没有发生过一样。服务器将断开客户应用程序原来的连接，并且认为客户端发生了崩溃。

处理这类通信中断的更好方法是在开发客户程序和服务器程序的时候考虑到对**带外**(out-of-band)数据发送的支持。带外数据是服务器在处理客户发送的其他所有数据之前必须处理的数据。一种解决方法是让客户将带外数据发送到一个另外的控制端点，服务器则监听该端点，同时也监听(以较低的优先级)常规数据传输的端点。另一种方法是通过发送原始请求所使用的连接来发送带外数据。比如在 TCP 中，传输紧急数据是可能的。当服务器接收到紧急数据时，服务器就被中断(比如通过 UNIX 系统中的信号机制)，随后服务器可以检查紧急数据并且根据数据类型来决定采取何种处理方式。

最后还有一个设计上的重要问题，就是服务器是否是**状态无关的**(stateless)。**状态无关服务器**(stateless server)不保存其客户的状态信息，而且也不将自身的状态变化告知任何客户(Birman 1996)。比如 Web 服务器就是状态无关的，它仅仅对输入的 HTTP 请求做出响应，这些请求包括向服务器上载文件或者更常见的从服务器获取文件，在处理请求的时候，Web 服务器彻底忽略了客户的存在。同样，Web 服务器所管理的一组文件(也许和文件服务器协作管理)的变化可以不必告知客户。

注意，在很多状态无关设计中，服务器实际是维护客户信息的，但重要的是，即使这些信息丢失，也不会打断所提供的服务。比如，Web 服务器通常记录所有客户请求，这对决定是否应该复制某些文档，和应该复制到哪很有用。显然，若日志丢失，顶多也就是服务器性能会受些影响，而不会有别的什么。

有一种特别的状态无关设计是服务器维护一种叫**软状态**（soft state）的信息。这种情况下，服务器会维持客户状态，但仅维持一小段时间。时间到期后就删除相关客户信息，回复到无状态行为。一个例子是服务器承诺会通知给客户一小段时间内的更新。在这之后客户就需要向服务器查询来获得更新。软状态的方法起源于计算机网络协议设计，但也能被用作服务器设计（Clark 1989；Lui 等 2004）。

相反，**状态相关服务器**（stateful server）一直保存客户端的信息直到显式地删除。这种服务器的典型例子是文件服务器，它让客户保留文件的本地副本，甚至进行更新操作。文件服务器维护一张表，表中条目是（客户，文件）对，服务器可以利用该表来跟踪目前哪些客户拥有对哪些文件的更新许可，并且跟踪哪里的文件的版本是最新的。

对于客户端来说，这种方法可以提升读写操作的性能。状态相关设计一个重要好处是相对于状态无关服务器的性能提升。然而，这个例子同时也体现了状态相关服务器的主要缺陷。即如果服务器崩溃，就必须恢复那些包含（客户端，文件）对条目的表，否则无法确保能够处理文件的最新版本。一般说来，状态相关服务器必须将自身的整个状态都恢复到崩溃之前。我们将在第 8 章中讨论，建立恢复机制将引入相当大的复杂性。如果采用状态无关设计，就不需要采取任何特殊措施来使崩溃的服务器恢复，只要简单地重新启动运行服务器，然后继续等待客户请求到达就行了。

（Ling 等 2004）认为应区分（临时的）**会话**（session）状态和永久状态。上面的例子是典型的会话状态的例子：此状态和一单个用户的一系列操作相联系，因此应该仅维持一小段时间，而不是长久。在涉及三方的客户-服务器结构中，会话状态通常被维持。在对请求客户做出响应前，应用程序服务器需要通过一系列查询来存取数据库服务器。在这个结构中，如果客户能重新发出原来的请求，则即使会话状态丢失，也不会有实际损害。因此可以用更简单和更不可靠的状态记录。

永久状态记录的通常是数据库中维持的信息，如客户信息、购买软件相联的密钥（key）等。然而，对大多数分布式系统来说，维持会话状态意味着状态相关设计，当系统失败时要求特殊措施，需要显式地假设服务器会长时间保持状态。当讨论容错时我们会详细考察这些问题。

在设计服务器的时候，无论选择采用状态无关设计还是状态相关设计都不应该对服务器所提供的服务造成影响。比如说，如果在读取文件或者写入文件之前必须打开该文件，那么状态无关服务器就必须用这样或那样的方法模拟这一打开文件的行为。通常的解决方法是，服务器响应对文件的读写请求时首先打开指定的文件，然后再执行实际的读写操作，最后立即关闭该文件。我们将在第 10 章中进行更为详细的讨论。

在某些情况下，服务器需要保留客户的活动记录，以便更加有效地响应客户的请求。比如说，Web 服务器有时可以立即将客户引导到该客户最常浏览的页面。只有在服务器拥有关于该客户的历史信息的情况下，这种方法才能够实施。如果服务器不能维持状态，一种常用的方法是让客户在发送请求的同时发送关于前一次访问的额外信息。在 Web 中，这种信

息常常由客户端浏览器透明地存储在 cookie 中，cookie 是一小段数据，其中包含有对服务器有用的针对特定客户的信息。浏览器本身永远不会执行 cookie，它只对 cookie 进行存储。

在客户首次访问服务器时，服务器将会把 cookie 和所请求的 Web 页面一起发送给浏览器，随后浏览器将会安全地将 cookie 隐藏起来。随后客户每次访问该服务器的时候，关于该服务器的 cookie 将会与请求一起发送给服务器。虽然原则上来说这种方法可以有效工作，但是浏览器将 cookie 送回服务器存储这个事实是对用户完全隐藏的，这就会引发许多隐私保护方面的问题。与奶奶做的 cookie（此处意为小甜饼——译者）不同，浏览器生成的 cookie 最好放在原处不动为妙。

3.4.2 服务器集群

在第 1 章我们简单讨论了**集群计算**（cluster computing），它是一种分布式系统。现在我们仔细考察一下服务器集群的组织及由它带来的设计问题。

3.4.2.1 常见的组织

简单说，服务器集群只是一组经网络连接的机器，每台机器运行一个或多个服务器。我们讨论的服务器集群是特指经局域网连接的机器，能提供高带宽和低延迟。

多数情况下，服务器集群逻辑上由三层（tier）组成，如图 3.12 所示。第一层是一个（逻辑上的）交换机，由它分配客户请求给服务器。交换机在功能上可变化很大。比如，传输层交换机接受 TCP 连接请求，再转发给集群中的某个服务器。我们下面还要讨论它。一个完全不同的例子是 Web 服务器。它接受 HTTP 请求，但只转发一部分给应用服务器作继续处理，然后收集结果并返回 HTTP 响应。

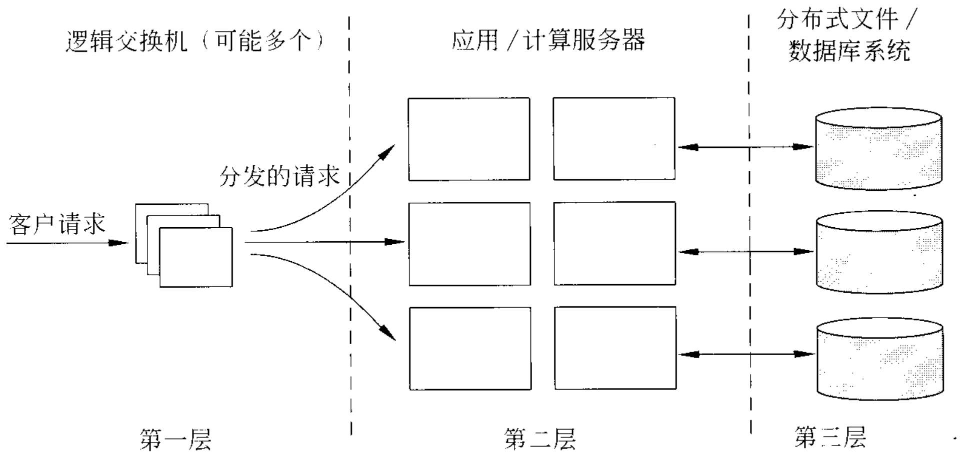


图 3.12 三层服务器集群通常的结构

就像所有的**多层**（multtiered）客户-服务器体系结构一样，很多服务器集群也包含了专用于应用处理的服务器。在集群计算中，这通常是运行在高性能硬件上专用于提供计算能力的服务器。然而，在企业服务器集群里，应用程序可能只需运行在相对低端的机器上，因为在这里的瓶颈不是计算能力而是数据存取。

这就出现了第三层，它由数据处理服务器组成，也就是文件和数据库服务器。按照服务器集群的使用目的，这些服务器可能运行在配置作为高速磁盘存取和拥有很大数据缓存的专用机器上。

当然，不是所有服务器集群都遵循这种严格的区分。经常是每台机器都有自己的本地存储，把应用和数据处理集成在单个服务器中，导致一种两层体系结构。比如，当通过服务器集群处理流媒体时，通常采用两层体系结构，每台机器作为专用的媒体服务器（Steinmetz 和 Nahrstedt 2004）。

在服务器集群提供多个服务情况下，不同机器可能运行不同的应用服务器。因此，交换机必须能够区分请求的服务，否则它不能正确地转发请求给适当的机器。很多第二层机器仅运行单个应用程序。这个限制来自于对适用的软件和硬件的依赖，也由于不同的应用程序常常由不同的管理员来管理。后者不喜欢干扰彼此的机器。

因此，我们可能会看到某些机器暂时空闲着，但别的机器却收到过载（overload）请求。这种情况下暂时迁移服务到空闲机器会管用。（Awadallah 和 Rosenblum 2004）提出的一种方法是用虚拟机，允许相对容易地迁移代码到实际机器上。在本章随后我们会再讨论代码迁移。

让我们仔细考察一下由交换机组成的第一层。一个重要的服务器集群设计目标是隐藏有多个服务器的事实。也就是说，运行在远程机器上的客户应用程序不应该需要知道集群的内部组织结构。这种存取透明性通过单个访问点来实现，由某种硬件交换机实现，比如一台专用机。

交换机形成了集群入口，提供唯一的网络地址。为了可扩展性和可用性，服务器集群可能有多个访问点，每个访问点由一独立的专用机器实现。我们仅考虑单访问点的情况。

一种标准的存取服务器集群的方式是建立一个 TCP 连接，在这之上应用级别的请求可作为一个会话的一部分来发送。通过撤除连接可结束会话。在传输层交换机的情况下，交换机接受到来的 TCP 连接请求，转发这些请求给一台服务器（Hung 等 1997；Pai 等 1998）。通常称为 TCP 转发的工作原理如图 3.13 所示。

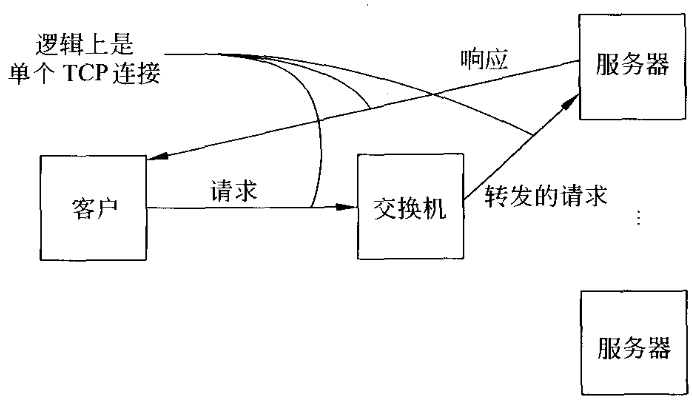


图 3.13 TCP 转发的原理

当交换机收到一个 TCP 连接请求时，它随即找到处理这个请求的最佳服务器，并转发这个请求包给这个服务器。服务器反过来会发送一个应答给请求的客户，但把交换机的 IP 地址插入到承载 TCP 数据段的 IP 包头的源地址域。注意，为了让客户继续执行 TCP 协

议，这种欺骗是必需的，因为客户在期待交换机的应答，而不是以前从没听说过的某个服务器。显然，实现 TCP 转发需要操作系统级别的修改。

我们已经看到，交换机对在多个服务器之间分配负荷能起重要作用。通过决定往哪转发请求，交换机也决定了哪个服务器来对请求作进一步处理。交换机可以用的最简单的负荷平衡策略是**轮转法**(round robin)：每次都从列表中选下一个服务器然后转发请求给它。

也可以用更高级的服务器选择策略。比如，假设集群提供多个服务。如果当请求到来时交换机能区分它们要求的服务，则交换机有更多信息来决定往哪转发请求。如果服务可以通过端口来区分，这种服务器选择可以仍然发生在传输层。再进一步的方法是让交换机实际检查到来请求的负载。只有在知道负载的结构时才可用这种方法。比如 Web 服务器的情况，交换机期待 HTTP 请求，基于这个请求的信息而决定谁来处理它。当我们在 12 章讨论基于 Web 的系统时还会再考察这些**内容感知的请求分布**(content-aware request distribution)。

3.4.2.2 分布式服务器

迄今为止讨论的服务器集群通常是相当静态地配置的。在这些集群中，通常有一个独立的管理机器用来跟踪可用的服务器，并适当地传递这些信息给别的机器，比如交换机。

我们提到绝大多数服务器集群提供单个访问点。当这个访问点失效时，无法使用集群。为消除掉这个隐患，可以提供几个访问点，公开它们的地址。比如，域名解析系统(Domain name system, DNS)能返回几个地址，所有的都对应于同一主机名。如果其中一个地址失效，这种方法还会要求客户重试几次。另外，它也没有解决要求静态访问点的问题。

无论从用户还是服务器的观点来说，稳定性，如长期有效的访问点，都是一个所希望的特性。从另一方面说，我们也希望在配置服务器集群(包括交换机)时有高度的灵活性。这导致了一种分布式服务器的设计。分布式服务器只是指可动态变化的一个机器群，它的访问点也可以变化，但对外却表现为一强有力的单台机器。(Szymaniak 等 2005)给出了这种分布式服务器的设计。这里我们简要描述它。

分布式服务器的基本思想是客户从一个可靠的、高性能的、稳定的服务器受益。高端主机通常可以提供这些属性，它们中的一些拥有引以为豪的超过 40 年的平均失败时间。然而，通过透明地组合简单的机器成为一个集群并且不依赖单台机器的可用性，有可能达到比单个组件更好的稳定性。比如，这样一个集群可从用户机器上动态地配置，就像在合作的分布式系统中那样。

让我们把注意力放在如何在这样一个系统中实现一个稳定访问点。其主要的思想是利用可用的网络服务，比如 IP 版本 6 的移动支持(MIPv6)。在 MIPv6 中，一个移动结点假设有一个**宿主网络**(home network)，通常呆在这个网络中并有稳定的地址，称为**宿主地址**(home address)。宿主网络有一特别的路由器，称为**宿主代理**(home agent)。当移动结点不在时，它将会处理送给移动结点的**网络包**(traffic)。当移动结点附着到一个外部网络时，它会收到一临时**需要的地址**(care of address，CoA)，其他结点可以发送网络包给这个地址。这个临时地址通告给宿主代理，它随后就可以转发送给移动结点的网络包到这个临时地址。注意，与移动结点通信的应用程序只会看到移动结点在宿主网络中的地址，而看不到那个临时地址。

这个原理可用来为分布式服务器提供稳定的地址。在这种情况下，给集群分配一个唯一的联系地址。联系地址是服务器生命周期内和外界通信的地址。任何时候分布式服务器都有一结点作为联系地址的访问点，这个角色可以容易地被另一结点取代。访问点在分布式服务器所在网络的宿主代理注册自己的地址为临时地址。这时所有的网络包都会导向访问点，它然后分配请求给当前参与分布式服务器的结点。若访问点失效，一个简单容错机制启动，另一个访问点会选出并注册一个新的临时地址。

这种简单的配置会使宿主代理和访问点成为潜在的瓶颈，因为所有的网络流量都要流经这两台机器。这种情形可以用 MIPv6 的一个称为**路径优化**(route optimization)的特性来避免。其如下工作。每当一个移动结点(宿主地址 HA)注册它的当前临时地址，记作 CA，宿主代理能转发 CA 给客户。后者将把（HA，CA）对存储在本地。从那时起，通信将直接送到 CA。尽管客户端的应用程序还能使用宿主地址，MIPv6 的底层支持软件会把这个地址翻译成 CA，然后实际通信都用 CA。

路径优化可以用来使不同的客户认为它们在和同一个服务器通信，而实际上这些客户是和分布式服务器的不同的成员结点通信的，如图 3.14 所示。为了做到这点，当分布式服务器的访问点转发客户 C1 来的一个请求给结点 S1(地址 CA1)时，它传递足够的信息给 S1，让 S1 启动路径优化过程从而让客户相信 S1 临时地址是 CA1。这将允许 C1 记下(HA，CA1)对。在这个过程中，C1 和 S1 的通信由访问点(还有宿主代理)来转发。这将使宿主代理相信临时地址没有变化，从而继续和访问点通信。

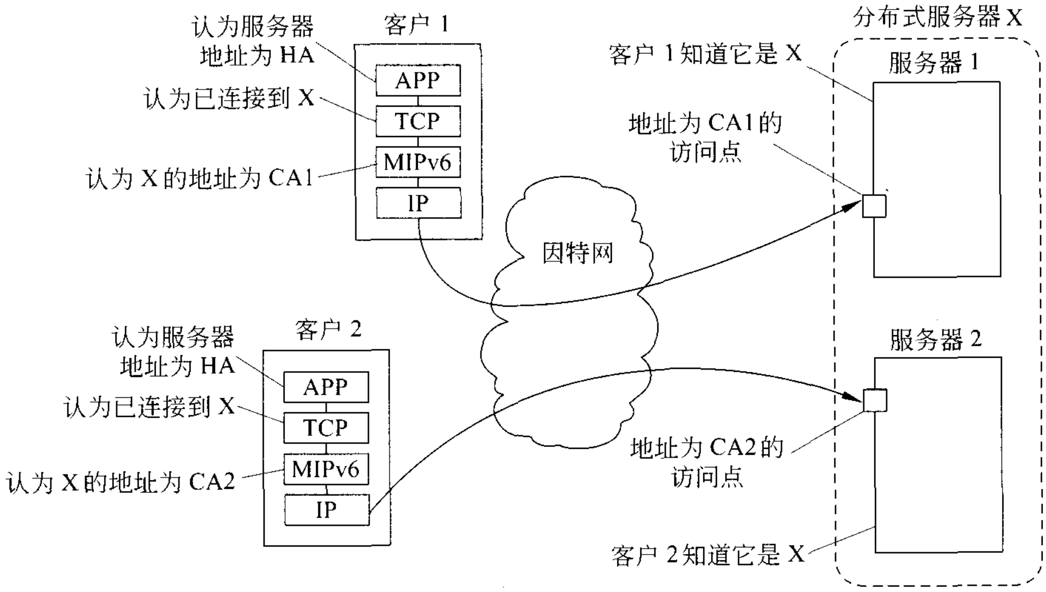


图 3.14 分布式服务器的路径优化

当然，路径优化的过程中，可能有别的客户的请求进来。这些请求会在访问点处于等待状态，直到它们可以被转发。比如说，从客户 C2 来的一个请求被转发给成员结点 S2(地址 CA2)，允许 S2 让客户记下(HA，CA2)对。因此，不同的客户将会直接和分布式服务器的不同的成员通信，而每个客户应用程序却觉得服务器地址只是 HA。宿主代理继续和联系地址上的访问点通信。

3.4.3 管理服务器集群

对外部来说，服务器集群应表现得像单台计算机。通常这也的确是事实。然而，当涉及到管理时，情况就很不一样了。下面我们讨论几个试图简化服务器集群管理的尝试。

3.4.3.1 通用方法

目前为止最通用的管理服务器集群的方法是把传统的单台计算机的管理功能扩展到服务器集群。最原始的做法是使管理员能从远程客户登录到集群的一个结点并执行本地管理命令来监控、安装和改变**组件**(component)。

稍微高级点的做法是不需要显式登录到一个结点，而是在一台管理机器上提供界面，允许从一个或多个服务器收集信息、升级组件、增减结点等。这种方法的主要好处是更容易提供集体操作(即面向一组服务器的操作)。它在实践上广泛应用，比如像 IBM 集群系统管理(Hochstetler 和 Beringer 2004)这样的管理软件。

然而，当集群增长到超过几十个结点时，这种管理方法就行不通了。很多数据中心需要管理几千台服务器，它们被组织成很多互相合作的集群。对这种应用使用集中式的管理服务器是不可能的。另外，容易看到大规模集群需要持续的维修管理(包括升级)。简单点说，如果 p 是一台服务器出错的概率，并假设服务器出错是独立的，则对一个 N 台服务器的集群，所有服务器都不出错的概率是$(1-p)^N$。若 p=0.001，N=1000，有 36%正常运行。

事实证明，对大型服务器集群的支持几乎无规则可循。有很多应该考虑的经验方法(Brewer 2001)，但没有系统性的方法来针对大规模的系统管理。尽管我们可以预期在取得更多经验后，上一章讲的自我管理方案最终会应用于这一领域，但集群管理现在还基本上处于起步阶段。

3.4.3.2 示例：PlanetLab

让我们仔细观察一个有点不寻常的集群服务器。PlanetLab 是一种协作的分布式系统。每个不同的组织捐献一台或多台计算机，加起来有几百台。这些计算机形成了一个单层的服务器集群，存取、处理和存储都发生在每个结点自己身上。出于需要，PlanetLab 的管理基本上是分布式的。在解释基本原理前，我们首先介绍主要的体系结构特点(Peterson 等 2005)。

在 PlanetLab 中，一个组织捐献一个或多个结点。一个结点可以简单地想象成一台计算机，虽然它也可以是另外一个集群。每个结点组织成如图 3.15 所示。有两个重要的组件(Bavier，2004)。第一个是虚拟机监视器(VMM)，它是一套增强的 Linux 操作系统。主要的增强是调整系统支持第二个组件，称为 Vserver。一个(Linux)Vserver 可以想象成是一个独立的环境，有一组进程在其上运行。不同 Vserver 上的进程完全独立。它们不能共享任何资源，比如文件、主存和网络连接，这与运行于同一操作系统上的进程不一样。一个 Vserver 提供一个由自己软件包、程序和网络设施组成的环境。比如，一个 Vserver 可以提供这样的环境：进程会注意到它能利用 Python 1.5.2 搭配老式的 Apache Web 服务器(如 httpd 1.3.1)。相反，另一 Vserver 可能支持最近版本的 Python 和 httpd。从这个意义说，

把Vserver称为服务器有点误导，由于它实际只是把分成组的进程分隔开。后面我们会简单地再讨论一下Vserver。

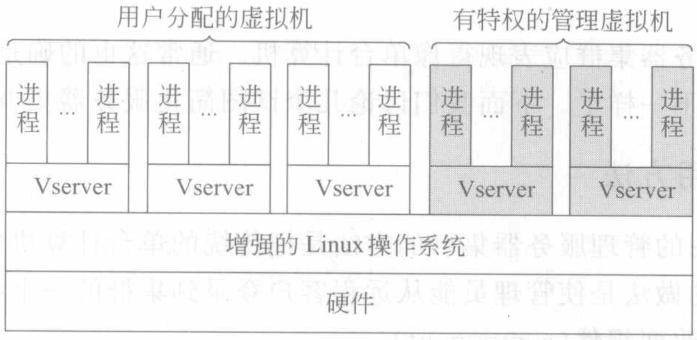


图3.15 PlanetLab结点的基本组织结构

Linux VMM确保Vserver之间是分离的：不同Vserver上的进程并发和独立地执行，每个进程只使用自己环境中可用的软件包和程序。不同Vserver上的进程被严格地隔离。比如，不同Vserver上的两个进程可以有相同的用户ID，但是这并不意味着它们属于同一用户。这种分离极大地简化了对来自不同组织的用户的支持，比如说，这些用户想要把PlanetLab用作testbed来试验完全不同的分布式系统和应用程序。

为支持这些试验，PlanetLab引入了切片(slice)的概念，它是一组Vserver，每个运行在不同的机器上。因此切片可被想成是Vserver集群，通过一组虚拟机来实现。PlanetLab的虚拟机运行在Linux操作系统上。此Linux系统已被扩展，增加了很多核心模块。

有几个问题让PlanetLab管理变得特殊。三个最显著的问题是：

(1) 结点属于不同的组织。应该允许每个组织规定谁可以在它们的结点上运行应用程序，并适当地限制资源的使用。

(2) 虽然有很多可用的监管工具，但它们都假设某种特定的软件和硬件组合。另外，它们都被调整成只在单个组织内使用。

(3) 来自不同切片但运行在同一结点上的程序不应互相干扰。这个问题类似于操作系统中的进程独立。

下面我们详细考察每个问题。

管理PlanetLab资源的工具叫结点管理器。每个结点有一个结点管理器，由一个Vserver实现。这个Vserver的所有任务就是创建这个结点上的别的Vserver并控制资源分配。结点管理器不做任何策略决定，它只是一个机制，用来提供让程序能在一个给定结点上运行的关键元素。

资源跟踪由资源规范（简称rspec）来实现。一个rspec标明一个时间段，这期间某些资源已被分配。资源包括磁盘空间、文件描述符、进入和出去的网络带宽、传输层端点、主存以及CPU使用。一个rspec由一个全局唯一的称为**资源能力**(resource capability, rcap)的128位的标识符来标识。给定一个rcap，结点管理器可以在本地表中搜索它标识的rspec。

资源是和切片结合的。也就是说，为使用资源，必需创建一个切片。每个切片和一个服务提供者相联系。服务提供者可看作是在PlanetLab上拥有账号的实体。每个切片可以通过一个(principal_id, slice_tag)对来标识。这里principal_id用来标识服务提供者，slice_tag是由服务提供者选择的标识符。

为创建一新切片，每个结点运行一个切片创建服务(SCS)，这将会联系结点管理器，请求它创建一个 Vserver 并分配资源。结点管理器不能通过网络直接联系，使它专心于本地资源管理。另一方面，SCS 不会接受所有人的切片创建请求。只有切片授权者(slice authorities)有资格请求创建切片。每个切片授权者都有对某一组结点的存取权限。最简单的模型是只有一个切片授权者，允许在所有结点上请求创建切片。

为了完成整个过程，服务提供者将联系一个切片授权者，并请求它在某一组结点上创建切片。服务提供者应该为切片授权者所信任，比如说，通过以前的认证和随后注册为 PlanetLab 用户。实践中，PlanetLab 用户通过基于 Web 的服务联系切片授权者。详情请见(Chun 和 Spalink 2003)。

这整个过程揭示了 PlanetLab 是通过中介管理的。这些中介里很重要的一类由切片授权者组成。切片授权者获得了在结点上创建切片的授权书。这些授权书通过带外获得，实质上是通过联系位于多个地方的管理员。显然这是一个费时的过程，不能由终端用户(或 PlanetLab 术语中的服务提供者)来完成。

除切片授权者外，还有**管理授权者**(management authority)，它负责管理结点。特别地，它确保它管理下的结点运行基本的 PlanetLab 软件并遵守 PlanetLab 设置的规则。服务提供者信任管理授权者管理下的结点具有良好的行为。

这种组织结构导致了如图 3.16 所示的用(Peterson 等 2005)中的信任关系所描述的管理结构。它们的关系如下：

(1) 结点拥有者把它的结点加入一个管理授权者的管理域，如果合适的话可能会限制使用方式。

(2) 管理授权者提供必需的软件用于增加一个结点给 PlanetLab。

(3) 服务提供者向管理授权者注册，信任它会提供良好行为的结点。

(4) 服务提供者联系切片授权者在一组结点上创建一个切片。

(5)切片授权者需要认证服务提供者。

(6) 结点拥有者为切片授权者提供切片创建服务用以创建切片。它实质上让切片授权者可以使用它的资源。

(7) 管理授权者把切片创建的任务委托给切片授权者。

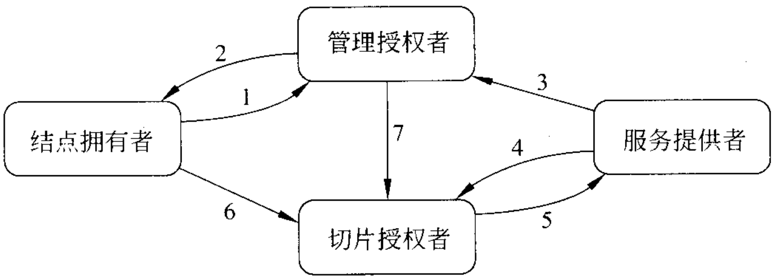


图 3.16 PlanetLab 实体间的管理关系

这些关系解决了以一种受控制的方式让结点可由别人使用的问题。在这种方式中，结点拥有者能依赖于一种良好的和安全的管理。在前面所述的三个问题中，第二个需要处理的问题是监管。需要一种统一的方法来允许用户观察在一个切片中，它们的程序运行是否良好。

PlanetLab 使用一个简单的方法。每个结点配置一组传感器，每个传感器能报告诸如 CPU 使用情况、磁盘活动等信息。传感器的复杂性可以随意，但重要的是它们总是报告基于每个结点的信息。这些信息可以通过一个 Web 服务器来存取：每个传感器都能通过简单的 HTTP 请求来存取（Bavier 等 2004）。

应该承认，这种方法还是相当原始，但它应被看作是高级监管方案的基础。比如，原理上说，没有原因使第 2 章中讨论的 Astrolabe 不能用作由多个结点的传感器数据合成的数据。

最后来讨论三个问题中的最后一个，也就是不同结点的程序的互相保护。PlanetLab 用 Linux 虚拟服务器（叫做 Vserver）来隔离切片。上面提过，Vserver 的主要思想是在它们自己的环境中运行应用程序，这包括通常在一个机器上共享的所有文件。通过 UNIX chroot 命令，这种隔离能相对容易做到。UNIX chroot 命令能有效改变文件系统的根，应用程序从这开始搜索文件。只有超级用户才能执行 chroot。

当然需要做更多的事情。Linux 虚拟服务器不仅分开了文件系统，而且通常共享进程的信息、网络地址、内存使用情况等。因此，一台物理机器实际上分成了多个单元，每个单元对应于一个完全独立的 Linux 环境，并与其他部分隔离开来。Linux 虚拟服务器的概述可在（Potzl 2005）中找到。

3.5 代码迁移

到此为止，我们所涉及的分布式系统所进行的通信仅限于传递数据。然而，在有些情况下需要传递程序（甚至可能需要传递正在执行中的程序），以简化分布式系统设计。在本节中，我们将详细考察代码迁移的实质。我们首先考察进行代码迁移的几种不同方案，随后讨论如何处理要迁移的程序所使用的本地资源，还要讨论一个特别难以解决的问题，即异构系统中的代码迁移。

3.5.1 代码迁移方案

在考察代码迁移的各种不同形式之前，让我们首先考虑一下代码迁移究竟有什么用处。

3.5.1.1 进行代码迁移的理由

传统上，分布式系统中的代码迁移是以**进程迁移**（process migration）的形式进行的，在这种形式下整个进程被从一台机器搬到另一台机器上去（Milojicic，2000）。将正在运行的进程移到另一台机器上去是一项复杂并且开销庞大的任务，所以要这么做必须有充分的理由。理由当然永远是性能上的考虑。其基本的思想是：如果把进程由负载较重的机器上转移到负载较轻的机器上去，就可以提升系统的整体性能。负载常常是用 CPU 队列长度或者 CPU 利用率来衡量的，也可以使用其他性能指标衡量。

负载分布算法决定了任务在一组处理器上的分配和重新分布。这种算法对于计算任务繁重的系统尤其有用。然而，在许多现代分布式系统中，对计算能力的优化不如通信量的减

少重要。此外,由于底层平台和计算机网络的异构性,代码迁移所能够带来的性能提升常常是基于定性推导得出的,而不是通过数学模型计算出来的。

举个例子,考虑一个客户-服务器系统,其中由服务器来管理一个巨型数据库。如果客户应用程序需要执行大量的涉及大量数据的数据库操作,最好先将客户应用程序的一部分搬运到服务器上运行,只通过网络发送结果。如果不这样做,服务器与客户之间传输的大量数据就会挤垮网络。在这种情况下,之所以要进行代码迁移是基于这样一个假定,那就是处理数据的地点离数据所在地点越近越好。

将服务器的一部分迁移到客户端的原因也是一样的。比如说,在许多交互式数据库应用程序中,客户必须填写表单,随后表单被转换成一系列数据库操作。比较好的做法是先在客户端处理表单,然后再把处理好的表单发送给服务器,这种做法可以避免在网络上传输大量的较小消息。这样客户就可以获得更好的性能,而同时服务器花费在处理和通信上的时间也会更少。

对于代码迁移的支持,还可以通过并行的使用来提升性能,而不会带来通常的并行编程所具有的复杂性。典型的例子是 Web 信息的搜索。以一种便于在站点之间移动的小移动程序(称为移动代理)的形式来实现这种搜索查询会相对简单。可以制作这种程序的多份副本,并且将每一份副本发送给不同的站点。与只使用单个程序示例相比较,这样做可以使速度得到线性提升。

除了可以提升性能之外,支持代码迁移还有其他方面的原因,其中最重要的是灵活性。构建分布式应用程序的传统方法是将应用程序分为不同部分,预先确定每一个部分应该在何处执行。依照这种方案执行会产生第 2 章中讨论过的各种多层客户-服务器应用程序。

然而,如果代码可以在不同机器之间移动,就可以动态地配置分布式系统。比如说,假定某个服务器实现了访问某个文件服务器的标准化接口。为了让远程客户访问文件服务器,服务器必须使用专门的协议。一般说来,基于这种专门协议的文件系统接口的客户端实现需要与客户端应用程序链接起来。这种方案要求在开发客户应用程序时就必须有可用的协议软件。

另一种方法是只在确实必要的时候让服务器提供客户的实现,也就是当客户绑定到服务器的时候。此时客户动态地下载该实现,经过某些必需的初始化步骤后激活该服务器。其原理如图 3.17 所示。这种将代码由远程站点动态移来的模型要求下载和初始化代码的协议是标准化的。同时还要求下载的代码能够在客户所在机器上执行。后面将在不同章节中讨论不同的解决方法。

这种动态下载客户端软件的显著优点在于,客户不需要预先安装与服务器通话所需的所有软件。可以在必要的时候将软件移来,同样也可以在不再需要该软件的时候将其丢弃。另一个优点是,只要接口是标准化的,就可以不受限制地改变客户-服务器协议及其实现,所做的改动并不会影响到现有的依赖于服务器的客户应用程序。当然,它也存在缺点,最严重的缺点我们将在第 9 章中进行讨论,那就是安全性方面的问题。盲目地信任下载代码实现的接口是极为危险的,它可能会访问您的毫无防备的硬盘,天知道它会将其中最敏感的信息发给什么人。

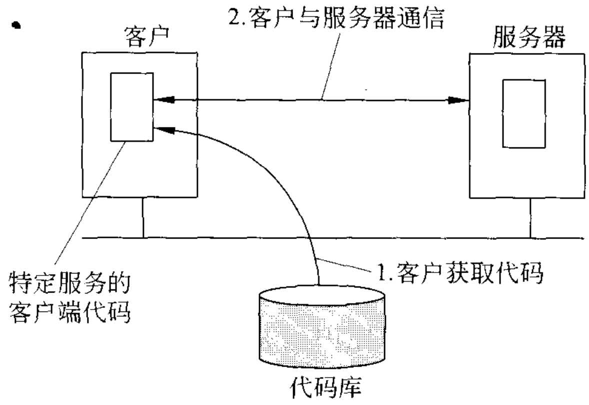


图 3.17　动态地配置与服务器通信的客户的原理。客户首先获取必要的软件，然后激活服务器

3.5.1.2　代码迁移模型

虽然代码迁移所说的只是在机器间移动代码，但是这个名词中包含的思想其实要丰富得多。传统上，分布式系统中的通信是关于进程间数据交换的。最广义的代码迁移可以涉及在机器间移动程序，目的是在目标机器上运行该程序。在某些情况下，与进程迁移相同，必须同时移动程序执行状态、未处理信号以及其他部分环境内容。

为了更好地理解代码迁移的不同模型，我们采用(Fugetta 等 1998)中提出的框架结构。在该框架结构中，进程包含如下 3 段。**代码段**(code segment)部分包含构成正在运行的程序的所有指令。**资源段**(resource segment)包含指向进程需要的外部资源的指针，这些外部资源包括文件、打印机、设备及其他进程等。最后是**执行段**(execution segment)，它用来存储进程的当前执行状态量，这些状态量包括私有数据、栈和程序计数器等。

代码迁移的最弱形式是仅仅提供**弱可移动性**(weak mobility)。在这种模型中，可以只传输代码段以及某些初始化数据。弱可移动性的典型特征是，传输过来的程序总是从预先定义的几个位置之一开始执行。Java 小程序(applet)就是这种情况，它总是从程序初始位置开始执行。这种方法的好处在于其简单性。弱可移动性只要求目标机器能够执行代码，这在本质上可以归结为使代码可移植。我们将在讨论异构系统中的迁移时继续讨论这些内容。

与弱可移动性相反，在支持**强可移动性**(strong mobility)的系统中，还可以传输执行段。强可移动性的典型特征是，可以先停止运行中的进程，然后将它移到另一台机器上去，再从刚才中断的位置继续执行。很明显，强可移动性要比弱可移动性强大得多，但是也更加难以实现。

可以通过是由发送者启动迁移还是接收者启动迁移来对迁移做进一步区分，这与可移动性的强弱无关。在**发送者启动**(sender-initiated)的迁移中，代码当前驻留在哪台机器上或者正在哪台机器上执行，就由该机器来启动迁移。一般来说，在往计算服务器上载程序时进行的就是发送者启动的迁移。通过因特网向 Web 数据库服务器发送搜索程序以在该服务器上进行查询，也是发送者启动迁移的一个例子。在**接收者启动**(receiver-initiated)的迁移中，代码迁移的主动权掌握在目标机器手中。Java 小程序就是这种迁移的一个例子。

接收者启动的迁移一般比发送者启动的迁移要更容易实现。在许多情况下，在客户和服务器之间进行的代码迁移的主动权由客户端掌握。如果要像发送者启动的迁移那样安全地向服务器上载代码，常常要求客户端预先在服务器上注册并通过验证。也就是说，要求服务器了解它的所有客户，这是因为客户很可能会访问服务器的资源，比如磁盘，而对这些资源的保护是极为重要的。相反，如果是接收者启动的迁移，代码下载工作一般可以用匿名进行。此外，服务器一般并不对客户资源感兴趣，而向客户端进行代码迁移只是为了提升客户端的性能。实际上，客户端只有有限的少数资源需要保护，比如内存和网络连接等。我们将在第 9 章中继续详细讨论代码迁移的安全问题。

在弱可移动性的情况下，迁移过去的代码是否由目标进程执行，还是重新启动一个单独的进程来执行，也存在区别。比如说，Java 小程序可以简单地由 Web 浏览器下载之后在浏览器的地址空间中执行。这种方法的好处是不需要启动单独的进程，也就避免了与目标机器进行通信，而主要缺陷是目标进程必须得到保护，以免在迁移来的代码的执行过程中受到有意或者无意的侵害。一种简单的解决方法是，由操作系统来进行管理，创建一个单独的进程来执行迁移来的代码。要注意，这种解决方法并没有解决刚才指出的资源访问方面的问题。这些问题还是需要处理。

如果不想直接移动运行中的进程，也就是说不进行进程迁移，那么可以通过**远程克隆**(remote cloning)的方式来支持强可移动性。与进程迁移不同，克隆将制作一份与原始进程完全相同的副本，但运行在不同的机器上。在 UNIX 系统中，远程克隆发生在派生出一个子进程并让它在远程机器上继续运行的情况下。克隆的优点是克隆得到的进程与原来已经在多个应用程序中使用的进程极为相似，唯一的不同点在于，克隆得到的进程是在另一台机器上运行。从这个意义上说，通过克隆来迁移是一种增强分布透明性的简单方法。

代码迁移的各种不同方法如图 3.18 所示。

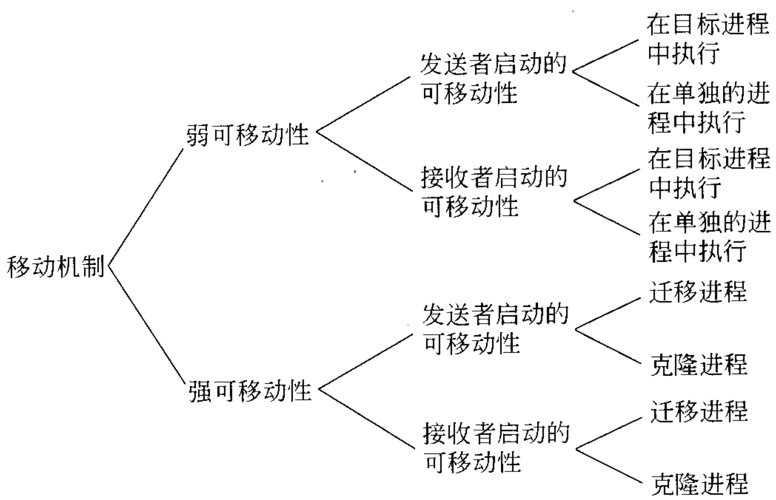


图 3.18 代码迁移的各种不同方法

3.5.2 迁移与本地资源

到此为止，我们只讨论了代码段和执行段的迁移。资源段迁移需要特别的关注。代码

迁移之所以如此困难，就是因为有时资源段在不作改动的情况下难以与其他段一同迁移。比如说，如果有一个进程中保存指向某个 TCP 端口的引用，通过该端口它可以与其他远端进程通信。该引用保存在进程的资源段中。当进程迁移到另一个地方时，它必须放弃使用该端口，然后在迁移到的地方请求一个新端口。而在其他情况下转移一个引用是不成问题的。比如说，如果一个进程中保存有以绝对 URL 的形式表示的文件引用，该进程无论驻留在哪一台机器上，该文件引用都有效。

为了理解代码迁移与资源段的关系，Fuggetta 等人将进程对资源的绑定方式分为三类。最强的绑定方式是进程使用资源的标识符来引用资源。在这种情况下，进程明确指定的只是要引用哪些资源。这种绑定称为**按标识符绑定**（binding by identifier），举个例子，进程使用 URL 来引用特定 Web 站点或者根据服务器所在的因特网地址来引用 FTP 服务器时就使用这种绑定。同样，使用指向本地通信终点的引用也会导致采用这种绑定方式。

进程对资源的绑定的一种较弱方式是，只使用资源的值。这种情况下，只要另一个资源能够提供一样的值，进程的执行就不会受到影响。依赖标准库的程序，比如那些使用 C 语言或者 Java 编程的程序，都是这种按值绑定（binding by value）的例子。本地必须保证这些库可用，但是这些库在本地文件系统中的确切位置依据站点不同可能会不同。为了保证进程能够正确执行，重要的不是用到了哪些文件，而是文件中的内容要正确。

进程对资源的绑定的最弱方式是，进程只指明它需要哪一种类型的资源。这种绑定叫做**按类型绑定**（binding by type），对诸如监视器、打印机等本地设备的引用都是按类型绑定的例子。

在迁移代码的时候，我们常常需要在不影响进程对资源的绑定的条件下改变指向资源的引用。改变引用的具体方式取决于资源能否与代码一起迁移到目标机器上去。更明确地说，需要就资源对机器的绑定方式进行考察，并区分为下列几种情况进行考虑。**未连接资源**（unattached resources）可以方便地在机器之间移动，这种资源一般是与要迁移的程序相关的数据文件。而**附着连接资源**（fastened resource）也可以进行复制或者移动，但是开销相对来说比较大。附着连接资源的例子包括本地数据库以及完整的 Web 站点。虽然这种资源在理论上并不依赖于当前的机器，但是将它们移动到另外的环境中去的做法通常并不可行。最后还有一种**紧固连接资源**（fixed resources），这种资源紧密地绑定到特定的机器或者环境，无法进行移动。本地设备一般是紧固连接资源。本地通信端点是另一个紧固连接资源的例子。

如果将三种类型的进程对资源的绑定与三种类型的资源对机器的绑定结合起来，就会产生 9 种组合形式，这些形式可供我们在考虑代码迁移时采用。在图 3.19 中列出了这些组合形式。

我们首先考虑一下进程能不能利用资源标识符来绑定到资源上。如果一个资源是未连接的，那么最好将它和要迁移的代码放在一起移动。然而，如果有其他进程共享该资源，就必须建立关于该资源的全局引用，即超越机器界限在全系统中都适用的引用。URL 是这种引用的一个例子。如果资源是附着的或者是紧固的，最好的解决方法还是建立全局引用。

意识到以下这一点是很重要的：建立全局引用并不仅仅是使用 URL，而且使用这种引用的开销有时昂贵得无法接受。举个例子，考虑一下某个供多媒体工作站使用的生成高画质图像的专用程序。实时制作高画质图像是一项非常繁重的计算任务，所以程序也许需要

		资源对机器绑定		
		未连接	附着连接	紧固连接
进程对资源绑定	按标识符	MV(或GR)	GR(或MV)	GR
	按值	CP(或MV,GR)	GR(或CP)	GR
	按类型	RB(或MV,CP)	RB(或GR,CP)	RB(或GR)

GR 创建一个系统范围的全局引用
MV 移动资源
CP 复制资源的值
RB 将进程重新绑定到本地可用资源

图3.19 在向另一台机器迁移代码时,根据引用本地资源方式的不同所应该采取的不同做法

移动到高性能计算服务器上去执行。建立多媒体工作站的全局引用意味着要在计算服务器和工作站之间建立连接通道。另外,为了满足图像传输的带宽需求,服务器和工作站还必须进行一些重要的处理工作。结果是将程序移动到计算服务器上的效果并不好,这主要是因为使用全局引用的开销太高了。

在以下这个例子中,建立全局引用也不是很容易的。如果要迁移的是使用本地通信端点的进程,就必须涉及对紧固连接资源的处理,进程是使用标识符绑定到该资源上的。这可以有两种解决方法。一种方法是,在进程迁移之后,在进程所在源机器上建立一个单独的进程并由迁移后的进程建立与源机器的连接,由单独的进程负责将所有传入消息通过连接转发给迁移后的进程。这种方法的主要缺点在于,无论何时只要源机器出错,与迁移后的进程的通信也就会失败。另一种方法是,让与迁移进程通信的所有进程都改变它们的全局引用,将消息都发送到目标机器上的新的通信终点。

如果是按值绑定,情况就不同了。先考虑紧固连接资源的情况。比如说,如果进程假定内存可由多个进程共享,则按值绑定会和紧固连接资源组合。在这种情况下,建立全局引用意味着需要实现分布式共享存储器机制。很明显,很多情况下这种解决方法并不真正可行和有效。

运行时库是典型的通过值引用的附着连接资源。一般说来,这种资源的副本在目标机器上是很容易得到的,也可以先将副本复制到机器上再进行代码迁移。如果要复制的数据量非常大,比如要复制文本处理系统中的字典或者词典,那么更好的做法是建立全局引用。

如果要处理的是未连接资源,情况最简单。只要没有多个进程共享该资源,最好的做法就是把资源复制(或者移动)到新地点。如果该资源被共享,那就只好建立全局引用了。

最后,还有按类型绑定。无论使用的资源对机器的绑定方式是何种类型,解决方法都是显而易见的,就是将进程重新绑定到本地可用的同一类型的资源上去。只有在本地没有同一类型的资源的情况下,才需要将原来的资源复制到新位置,或者建立全局引用。

3.5.3 异构系统中的代码迁移

到此为止,我们都默认这样一个假定,那就是迁移后的代码能够在目标机器上很好地执行。如果是同构系统,这个假定是成立的。然而,分布式系统通常是建立在一组异构平台的基础上的,这些平台各自的操作系统和机器体系结构都不同。在这种系统内的迁移要求每

一个平台得到支持，也就是说代码段必须能够在所有平台上执行。同时，必须确保能在所有平台上正确表示执行段。

异构性所导致的问题在很多方面与可移性相同，解决方法也非常相近，对此并不应该感到惊奇。比如说，在19世纪70年代末，产生了一种可以解决Pascal在不同机器间移植时大部分问题的简单方法，就是生成用于抽象虚拟机的与机器无关的中间代码(Barren 1981)。当然，该虚拟机需要在多种平台上实现，这样就可以允许Pascal程序在各处运行。虽然这种简单的思想已经沿用好多年了，但它从未真正成为解决其他语言（特别是C语言）的移植性问题的通用流行方法。

大约25年以后，异构系统中代码迁移的解决方法受到脚本语言及诸如Java之类的高度可移植语言的冲击。这类解决方法的共同点是它们都依赖于虚拟机，该虚拟机要么直接解释源代码（比如脚本语言就是这样），要么就解释由编译器生成的中间代码(Java就是这样)。可见，对于语言开发者来说，选择正确的时间和在正确的领域进行开发也是很重要的。

近来的工作已开始弱化对编程语言的依赖，特别地，有些提出的方法不仅迁移进程，而且迁移整个计算环境。基本思想是拆分综合环境并给予进程对自己计算环境的同样的观感。

如果适当地折分，有可能把底层系统的一部分分离出来并把它迁移到别的机器。用这种方法，进程迁移将实际上提供某种形式的强可移动性，因为它们在执行中的任何一点都可以被迁移，并且迁移完后可以从中断的地方恢复运行。另外，这也解决了当进程有对本地资源的绑定时被迁移的很多复杂问题，因为很多情况下这些绑定得到保留。也就是说，本地资源常常是作为被迁移的环境的一部分。

有几个原因促使迁移整个环境，但最重要的是：当一台机器需要关机时，迁移整个环境允许进程继续运行。比如，在一个服务器集群中，系统管理可能决定关闭或换掉一台机器，但不一定需要终止所有进程。相反，它可以暂时冻结环境，移到另一机器上（在这里被冻结的环境会和现在的环境共存），并解冻被冻结的环境。显然，这是非常强有力的管理长时间运行的计算环境及其进程的方法。

我们考虑一个(Clark 2005)中讨论的迁移虚拟机的特例。这种例子中，作者专注于虚拟化操作系统的实时迁移。虚拟化操作系统给通过一个共享局域网实现紧密耦合的服务器集群提供了方便。这种情况下，迁移涉及到两个主要问题：迁移整个内存映像和迁移对本地资源的绑定。

对第一个问题，原则上有三种方法来处理迁移（可以合在一起）：

(1) 把内存页面送给新机器并重发在迁移过程中修改了的页面。

(2) 终止当前虚拟机，迁移内存，并启动新虚拟机。

(3) 让新虚拟机按需要抽取新页面，也就是说，在新虚拟机立刻启动进程并按需要复制内存页面。

如果迁移的虚拟机在运行活动的服务，也就是说提供持续的服务，第二个选项可能导致不可接受的长停机时间。另一方面，第三个选项完全基于需要的方法可能会极大地延长迁移时间，而且可能也会导致很差的性能，因为把迁移进程的工作集移到新机器很化时间。

作为一个替代方案，(Clark 等 2005)提出使用一个预复制的方法，它结合了第一个选项和第二个选项中的简短的停止-复制阶段。这种组合可以降低服务故障时间到200毫秒或

更少。

至于本地资源，当处理集群服务器时事情变简单了。第一，因为只有一个网络，我们只需要在网络上公布新的网络地址到MAC地址的绑定，因而客户可在正确的网络地址联系迁移了的进程。最后，如果能假设存储是作为分开的一方(像图3.12所示)，则迁移对文件的绑定类似地简单。

综合效果是，不仅是进程，我们现在实际上看到连整个操作系统都可以在机器之间迁移。

本章小结

在分布式系统中，进程是一个基本的部分，它们构成了不同机器间通信的基础。一个重要的问题是，进程内部的组织结构究竟是什么样的，它们是否支持多个控制线程。分布式系统中的线程对于在执行阻塞性I/O操作时继续使用CPU是非常有用的。如果采取了多线程方式，就可以构建效率更高的服务器，让服务器并行地运行多个线程，其中某几个线程可以由于阻塞而处于等待状态，直到磁盘I/O或者网络通信完成为止。

以客户端-服务器的方式来组织分布式应用程序已经证实是很有用的。客户进程一般实现用户接口，这种用户接口可以只提供简单的显示，也可以提供能够处理复合文档的高级接口。而且客户软件还通过隐藏与服务器通信的细节，包括服务器当前所在位置和服务器是否被复制等细节，来取得更好的分布透明性。另外客户端软件还可以部分地负责隐藏故障及恢复工作。

服务器一般要比客户更加复杂，但是与设计相关的问题相对较少。比如说，服务器可以是迭代的也可以是并发的，可以实现一种服务也可以实现多种服务，可以是状态无关的也可以是状态相关的。其他设计方面的问题涉及寻址服务，以及在服务请求已经提出甚至已经在处理过程中的情况下中断该服务的机制。

把很多服务器组织成一个集群需要特别关注。一个通常的目标是对外部世界隐藏集群的内部细节。这意味着集群的组织应该和应用程序无关。为达此目的，大多数集群使用一个**单访问点**(entry point)把消息转发给集群中的服务器。一个挑战性的问题是透明地把这个单访问点换成一个完全分布式的解决方案。

在不同机器间进行代码迁移是分布式系统中的一个重要主题。支持代码迁移的理由是为了提高性能及灵活性。当通信的开销很大时，我们有时可以把计算工作由服务器移到客户端进行，让客户尽可能多地进行本地处理，以减少通信量。如果客户能够动态下载与特定服务器通信所需的软件，就可以提高灵活性。从服务器下载的软件是该服务器专用的，而不需要客户预装该软件。

代码迁移会带来与本地资源使用相关的问题，因为迁移时要求资源同时迁移，并且在目标机器上重新绑定到本地资源，或者使用系统范围的网络引用。另一个问题是，在迁移代码时要考虑到异构性。目前的实践证明，最好的解决方法是使用虚拟机来处理异构性。虚拟机可以采用像Java中的进程虚拟机的形式，或使用虚拟机监视器来允许一组进程随同底下的操作系统一起迁移。

习 题

1. 比较使用单线程文件服务器读取文件和使用多线程服务器读取文件有什么不同。花费15ms 来接收请求、调度该请求并且完成其他必须的处理工作，假定需要的数据存放在主存储器的缓存中。如果需要磁盘操作，就需要额外多花 75ms，在磁盘操作的过程中线程处于睡眠状态。如果服务器采用单线程的话，它每秒能处理多少个请求？如果采用多线程呢？
2. 对服务器进程中的线程数目进行限制有意义吗？
3. 在文中我们描述了一个多线程的文件服务器，说明了为什么它比单线程服务器和有限状态机服务器更好。有没有这样的环境，在其中使用单线程服务器会更好？给出这种环境的例子。
4. 将轻量级进程与单个线程静态关联起来并不好，为什么？
5. 如果每个进程只使用单个轻量级进程也不好，为什么？
6. 描述一种使用与可运行线程数目相等的轻量级进程的方法。
7. X 指定用户终端作为服务器，而把应用程序称为客户。这样做可以吗？
8. 协议 X 的可扩展性不怎么好。如何处理这个问题？
9. 代理可以通过调用所有副本来支持引用透明性。能否对（服务器端的）对像的副本进行调用？
10. 通过生成进程来构建并发服务器与使用多线程服务器相比有优点也有缺点。给出部分优点和缺点。
11. 粗略地设想一种多线程服务器的设计，该服务器必须使用套接字作为面对底层操作系统的传输级接口，以支持多种协议。
12. 如何防止应用程序绕过窗口管理器破坏屏幕显示？
13. 维护到客户的 TCP/IP 连接的服务器是状态相关的还是状态无关的？
14. 想象一下，某个 Web 服务器维护一个列表，该列表中的内容是 IP 地址与该地址最近访问过的 Web 页的映射关系。当一个客户连接到该服务器的时候，该服务器在列表中查找该客户，如果找到的话就返回注册过的页面。这个服务器是状态相关的还是状态无关的？
15. 在 UNIX 系统中，可以通过让进程在远程机器上派生出一个子进程来支持强可移动性。请说明这种机制的工作机理。
16. 图 3.18 指出，强可移动性不能与在目标进程中已迁移代码的执行结合在一起。请举出一个反例。
17. 考虑某个进程 P，它请求访问与自己位于同一台机器上的本地文件 F。在 P 迁移到另一台机器上以后，它还需要访问 F。如果文件对机器的绑定是紧固的，如何实现对 F 的系统级引用？
18. 详细描述在 TCP 交接时 TCP 包的流向，包括 TCP 包头中的源地址和目的地址信息。

第4章

通信

进程间通信是所有分布式系统的核心。如果不仔细分析分布式系统中各机器间信息交换的机制,对分布式系统的研究就成了空谈。分布式系统中的通信都是基于底层网络提供的低层消息传递机制的。就像我们在第 1 章里谈到的那样,通过消息传递来描述通信过程比使用基于共享存储器的原语来描述要更困难。现代分布式系统中常常含有数千甚至数百万个进程,这些进程分散在诸如因特网的不可靠网络中。除非用其他的技术来替代计算机网络的原始通信功能,否则要开发大规模分布式应用程序是极为困难的。

在本章的开头,我们将讨论进行通信的进程必须遵守的规则(这种规则称为协议),并且把讨论重点放在协议的分层结构上。然后将讨论 4 个广泛使用的通信模型:远程过程调用(remote procedure call,RPC)、远程方法调用(remote method invocation,RMI)、面向消息的中间件(message-oriented middleware,MOM)以及流(stream)。本章还讨论了发送数据到多个接收器(称为多播)的常见问题。

首先要讨论的是分布式系统中的远程过程调用(RPC)通信模型。RPC 的目的在于将消息传递的大部分复杂性隐藏起来,它比较适用于客户-服务器应用程序。

在许多分布式应用程序中,通信并不遵循严格的客户-服务器交互模式。在这种情况下,事实证明,从消息的角度来考虑会更恰当。然而,计算机网络的低层通信功能由于缺乏分布透明性,从许多方面来说并不适用。另一种方法是使用高层消息队列模型,在这种模型中,进行通信的方式与电子邮件系统的通信方式非常类似。因此面向消息的中间件(MOM)是一个重要的主题,值得用专门的一节来讨论它。

随着多媒体分布式系统的出现,很多系统都缺乏对诸如音频和视频这样的连续媒体的通信的支持。因此有必要引入**流**(stream)的概念。流可以在有时间限制的条件下支持消息的连续流。我们将以单独的一节对流进行讨论。

最后,随着我们对多播的理解,就可以了解数据通信的更好解决方法了。在本章的最后一节将来关注这个内容。

4.1 基础知识

在开始讨论分布式系统的通信之前,首先让我们来重温有关通信的一些基础知识。接下来的一节将简要地讨论网络通信协议,因为这些是构成所有分布式系统的基础。这里,我

们采用了一种与以往不同的方法，对分布式系统中出现的不同通信类型进行分类。

4.1.1 分层协议

由于没有共享存储器，分布式系统中的所有通信都是基于（低层）消息交换的。如果进程A要与进程B通信，A必须首先在自己的地址空间中生成该消息，再执行一个系统调用，通知操作系统将该消息通过网络发送给B。虽然这个基本思路听起来很简单，但是做起来并不简单，为了避免混乱，A和B必须先就传输的所有位所代表的含义达成一致。如果A发送了一部精彩的法文新小说，采用的是IBM的EBCDIC字符编码方式，而B等待接收的却是一张采用ASCII编码的超级市场的英文存货清单，显然，这样的通信是毫无用处的。

需要制定多种这样的**协定**（agreement）。比如，用多高的电压来表示0（低电平）信号，又用多高的电压来表示1（高电平）信号？消息接收方如何知道用哪一位标志消息的结尾？如何检测消息是否丢失或者损坏，发现后该怎么办？数值、字符串以及其他数据项的长度是多少，表示方式是什么？简而言之，必须在不同层次上制定多种协定，其中包括从位传输的低层细节到信息表示的高层细节在内的各层。

为了更加方便地对通信中涉及的多个不同层次进行处理，并解决其中存在的问题，**国际标准化组织**（ISO）颁布了一个参考模型，该模型清楚地标明了通信涉及的各个层次，为这些层次给出了标准名称，并且指出各层执行的特定任务。该模型称为**开放式系统互连参考模型**（open systems interconnection reference model）（Day和Zimmerman 1983），通常简称为ISO OSI模型，有时就称为OSI模型。需要强调指出的是，作为OSI模型的一部分而开发的协议从未得到过广泛应用。然而，学习底层模型本身对于理解计算机网络是十分有用的。虽然我们并不想在这里给出该模型的完整描述以及它涉及的所有内容，但是进行简短的讨论还是很有好处的。如果想知道关于该模型的更多细节，请参见（Tanenbaum 2003）。

OSI模型是设计用来支持开放式系统间的通信的。所谓开放式系统是准备通过一系列标准规则来与其他开放式系统通信的系统，这些规则规定了发送和接收的消息的格式、内容以及相应的含义。对这些规则进行归纳总结，加以形式化，就形成了**协议**（protocol）。为了使一组计算机能够通过网络相互通信，它们必须使用相同的协议。可以把协议划分为两大类。一类是**面向连接**（connection oriented）的协议，使用这种协议，消息发送方和接收方必须首先显式地确立连接，可能还需要就采用的协议进行协商，然后两者才能进行数据交换。在通信完毕之后，它们必须释放（终止）连接。电话系统就是一个面向连接的通信系统。另一类是**无连接**（connectionless）协议，使用这种协议，交换数据之前不需要有建立连接的过程，消息发送方只需要在准备好的时候开始传送第一个消息即可。将信件投入邮箱就是无连接通信的一个例子。对于计算机通信来说，无论面向连接的通信还是无连接的通信都是很常用的。

在OSI模型中，通信过程划分为7级（层），如图4.1所示。每一层负责处理通信中某个特定方面的问题。这样就可以把要解决的问题划分成多个易于处理的部分，每个部分都可以相对独立地进行处理。每一层都规定了与上面一层之间的接口，接口中包含一组操作，这些操作共同定义了该层向其用户提供的服务。

如果机器1上的进程A想与机器2上的进程B通信，它先生成一个消息，然后将该消

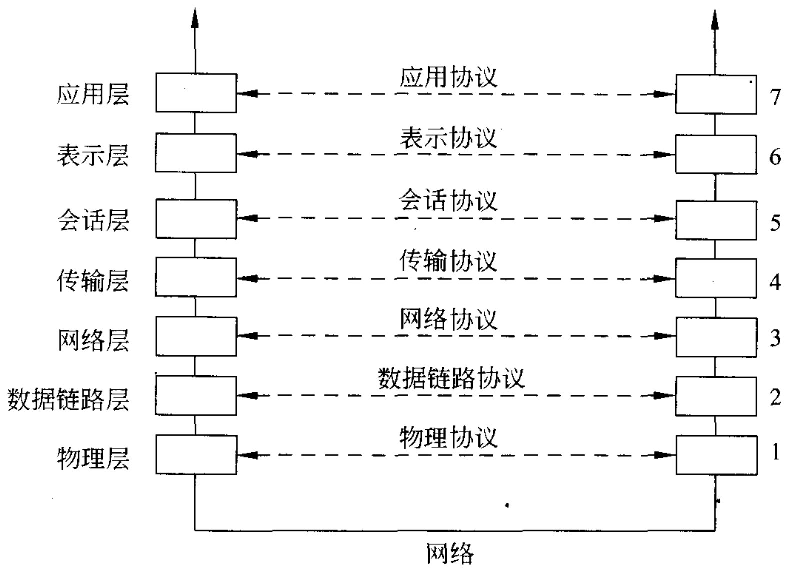


图 4.1 OSI 模型中的层、接口以及协议

息传递给机器 1 上的应用层。该层可能是一个库过程,也可能通过其他方式实现(比如在操作系统内部实现,或者由外部网络处理器实现)。应用层软件随后在消息前面添加一个**报头**(header),并通过第 6 层与第 7 层之间的接口将处理后的消息传递给表示层。表示层在得到的消息中添加自己的报头,并将得到的消息传给下一层即会话层。依此类推,逐层向下传递。某些层不仅要在消息开头添加自己的报头,而且还要在消息末尾添加**报尾**(trailer)。当消息最终到达底层物理层时,由该层执行实际的消息传输,即把它放置到物理传输介质上去。此时的消息如图 4.2 所示。

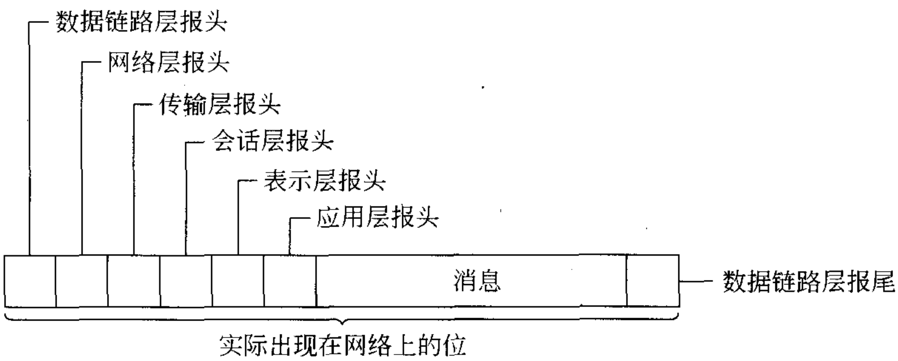


图 4.2 在网络上传输的典型消息

当消息送达机器 2 时,又被逐层向上传递,每一层对自身的报头进行剥离和检验。最后,消息抵达接收方,即进程 B,进程 B 可使用相反的路径进行回复。第 n 层添加的报头中的信息是供第 n 层协议使用的。

以下这个例子说明了分层协议的重要性。我们来考虑两家公司之间的通信问题,这两家公司分别是 Zippy 航空公司以及它的服务商 Mushy 餐饮国际有限公司。每个月,Zippy 公司的客运服务负责人都会打发她的秘书与 Mushy 公司的销售经理秘书进行联系,订购 10 万份鸡肉。按照传统的做法,订单应该通过邮局来递送,然而,由于邮政服务很糟糕,两位秘书决定放弃这种方式,转而采用传真方式进行通信。采用哪种通信方式并不需要征得

他们各自老板的同意，因为他们之间达成的协议只涉及订单的物理传输方式，而与订单的内容无关。

同样，客运服务负责人也可以决定不再订购鸡肉，而订购 Mushy 公司新推出的羊上肋肉，他所做出的这个决定并不会对秘书的工作造成影响。要注意的是，这里分为两层，即老板和秘书。每一层都有自己的协议（通话的主题和采用的技术），可以单独改变某一层的协议，而不会对另一层造成影响。分层协议的可贵之处正是在于它带来的这种独立性。随着技术的发展，可以对任何层单独进行改进，而不会对其他层造成影响。

从图 4.1 中可以看出，在 OSI 模型中不是两层，而是七层。特定系统中使用的协议集称为**协议簇**（protocol suite）或者**协议栈**（protocol stack）。把参考模型与其实际的协议区分开来是很重要的。我们曾经指出，OSI 协议从未流行过。相反，得到广泛应用的是那些为因特网开发的协议，比如 TCP 协议和 IP 协议。在以下各节中，我们将从底层开始，依次对 OSI 各层进行简要分析。然而，我们并不给出 OSI 协议的例子，而是在适当的地方指出各层中使用的某些 Internet 协议。

4.1.1.1 低层协议

我们先讨论 OSI 协议组中的三个最低层协议。这三层共同实现了计算机网络的基本功能。

物理层负责对 0（低电平）信号和 1（高电平）信号的传输。物理层中的关键问题是 0 和 1 信号分别使用多高的电压、每秒传输多少位以及是否可以同时进行双向传输等。另外，该层还关心网络链接器（插头）的尺寸和形状，以及插头上针脚的个数和各引脚的作用。

物理层协议制定电气、机械和信号方面的标准，以确保某台机器发送的一个“0”位在接收机器上仍然被识别为“0”而不是“1”。已经制定了许多（在不同介质中使用的）物理层标准，其中包括供串行通信线路使用的 RS-232-C 标准。

物理层只负责传输位。只要没有错误发生，就不会有任何问题。然而，实际的通信网络中是容易发生错误的，所以需要采用某种机制来检测并且纠正这些错误。实现这种机制是数据链路层的主要任务。该层将若干位组成一个称为**帧**（frame）的单元，并检查每一帧是否被接收方正确接收。

为了完成这些工作，数据链路层在每一帧的开头和结尾分别放置特殊的位模式来对头尾作出标记，并且用某种方法将帧中所有的字节相加，计算出**校验和**（checksum）。数据链路层将得到的校验和附加到帧中。当该帧送达的时候，接收方对数据校验和进行重新计算，并将得出的结果与该帧后跟的校验和进行比较。如果两者一致，说明该帧传输过程中没有发生错误并接受该帧。如果两者不一致，接收方就要求发送方重发该帧。每一帧都带有一个序列号（在其报头中），所以可以很容易地对不同帧进行区分。

在局域网中，发送方一般并不需要了解接收方所在的位置，它只须将消息放到网络上，然后接收方会取走该消息。然而，在广域网中情况就不同了。广域网由大量计算机组成，其中每台机器都拥有连接到其他机器的若干条线路，就像一幅显示大城市以及大城市之间相连的道路的大范围地图。消息从发送方传送到接收方的过程中，可能要经过好多**跳**（hop），在每个跳中，都要选择一条传出线路。如何选择最佳路径的问题称为**路由**（routing），这是网络层的首要任务。

最短的路线并不总是最佳路线，这就使得路由问题很复杂。真正最重要的是所选择路径的传输延迟时间，而这又与各条线路上的网络通信流量以及排队等候发送的消息数量有关。因此，传输延迟会随时间的推移而发生变化。一些路由算法试图根据动态负载来选择路由，而其他的算法则只是根据一段较长时间内的负载均值来进行决策。

目前应用最广泛的网络协议可能就是无连接的**网际协议**(Internet Protocol, IP)，它是因特网协议簇的一部分。不需要先建立连接就可以发送 IP **包**(packet)(包是表示网络中的消息的技术术语)。每个 IP 包彼此独立地发送到目的地，而且在包内部不指定或记录路径。

4.1.1.2 传输协议

有一些服务对于构建网络应用程序是必需的，但在网络层接口中却没有提供，这些服务由传输层来实现。从这个意义上说，传输层构成了基本网络协议栈的最后一部分，也就是说，传输层将底层网络转换成可供应用程序开发人员使用的形式。

发送方发给接收方的包有可能在途中丢失。虽然某些应用程序可能拥有自定义的错误恢复手段来进行处理，但是多数程序还是希望连接是可靠的。传输层的任务就是提供这种可靠的连接服务。这样，应用层在把消息交付给传输层的时候，消息将会按预期的那样被无损地送达给接收方。

当从应用层接收到消息时，传输层将消息分为适于传输的足够小的块，每一块都分配一个序列号，然后将它们发送出去。在传输层报头中涉及的内容包括哪些包已发送、哪些包已接收到、接收方所拥有的空间还能够容纳多少包、哪些包应该重发以及其他一些类似的主题。

可靠的传输连接(根据定义，是面向连接的)既可以建立在面向连接的网络服务之上，也可以建立在无连接的网络服务之上。如果是前者，所有的包都将依发送时的次序到达接收方(如果所有的包都能安全到达的话)；而如果是后者，由于各包经过的路线可能不同，后发送的包有可能先到达。由传输层中的软件负责将所有的包依照原来的顺序排列好，以维持传输连接就像一根大管子的假象：将消息从一端放入，随后从另一端按照放入时的顺序将消息无损地提取出来。提供这种点对点的通信方式是传输层的重要特征之一。

因特网传输协议称为**传输控制协议**(transmission control protocol, TCP)，详细描述见(Comer 2006)。TCP 与 IP 结合成的 TCP/IP 协议是目前网络通信中的事实标准。因特网协议簇也支持无连接的传输协议，该协议称为**通用数据报协议**(universal datagram protocol, UDP)，从本质上说它只不过是附加了某些次要内容的 IP 协议。不需要面向连接协议的用户程序通常使用 UDP。

其他传输协议都是为了某种专门用途而制定的。例如，为了支持实时数据传输，制定了**实时传输协议**(real-time transport protocol, RTP)。RTP 是一种框架性协议，也就是说，它规定了实时数据所用的包的格式，但是并没有提供确保数据成功送达的实际机制。另外，它还描述了另一种协议，用于监视和控制 RTP 包的数据传输(Schuizrinne 等 2003)。

4.1.1.3 高层协议

在传输层之上，OSI 又划分了另外三个层。在实践中，只用到其中的应用层。事实上，在因特网协议簇中，传输层之上的所有内容都合到一起了。在本节中我们将会看到，对于中

间件系统来说，OSI 方案和因特网方案都不真正适用。

会话层本质上是传输层的增强版本，它提供对话控制，用于跟踪正在谈话的是哪一方，它还提供了同步功能。同步功能对于长时间的传输过程非常有用，它允许用户在传输过程中插入若干检验点，这样如果传输崩溃，只需要回退到最近的一个检验点，而不需要从头开始重新传输。在实践中，应用程序很少用到会话层，几乎不对它提供支持。在因特网协议簇中没有与会话层对应的层。

低层主要考虑的问题是如何将要传输的所有位从发送方可靠而高效地传送给接收方，而在表示层中主要考虑的则是这些位所表示的意义。大多数消息并不是由随机的位串组成的，而是由更结构化的信息，比如人名、地址、金额等组成。在表示层中，有可能定义包含这种结构化信息字段的记录，随后由发送方通知接收方消息中包含有符合某种特定格式的记录。这种做法可以使得在内部数据表示方法不同的机器之间的通信更加方便。

OSI 应用层起初是准备用来容纳一组标准网络应用程序的，比如那些供电子邮件、文件传输以及终端仿真使用的应用程序。但是现在，应用层变成了所有由于各种原因不适于归纳到某个较低层中去的应用程序和协议的容器。从 OSI 参考模型的角度来看，几乎所有的分布式系统都只是应用程序而已。

这个模型中缺乏对应用程序、针对特定应用程序的协议以及通用协议的明确区分。例如，因特网的**文件传输协议**（File Transfer Protocol，FTP）（Postel 和 Reynolds，985；Horowitz 和 Lunt 1997）定义了一种在客户和服务器间传输文件的协议。该协议不应与 ftp 程序相混淆。ftp 程序是指一种供最终用户使用的应用程序，用来传输文件，而且这种程序恰好（当然不完全是偶然的巧合）实现了因特网 FTP 协议。

另一个关于针对特定应用程序的协议的例子是**超文本传输协议**（HyperText Transfer Protocol，HTTP）（Fielding 等 1999），它是设计用来对 Web 页面进行远程管理和处理的。该协议由 Web 浏览器以及 Web 服务器这样的应用程序实现。然而，HTTP 目前也用在与 Web 没有直接联系的系统中。例如，Java 的对象激活机制就使用 HTTP 来向受防火墙保护的远程对象发出调用请求（Sun Microsystems 2004b）。

还有为数不少的对多数应用程序十分有用的通用协议，但是这些协议并不能算是传输协议。在许多情况下，这种协议只能归入中间件协议的范畴。下面来讨论中间件协议。

4.1.1.4 中间件协议

中间件是一种应用程序，它在逻辑上位于应用层中，但在其中包含有多种通用协议，这些协议代表各自所在的层，独立于其他更加特别的应用。可以在高层通信协议和用于建立各种中间件服务的协议间作出区分。

有多种协议可以对各种中间件服务提供支持。比如说，有很多方法可以进行**认证**（authentication）——为声称的身份提供证明，我们将在第 9 章中对认证进行讨论。认证协议与任何特定的应用程序都没有紧密的联系，但是可以作为一种通用服务集成到中间件系统中。同样，授权协议（这种协议允许通过验证的用户和进程对其拥有授权的资源进行访问）本质上也可以是通用的、与应用程序无关的。

作为另一个例子，我们来考虑一下第 8 章中的几种分布式提交协议。提交协议规定，在一组进程中，某种操作要么由全部进程共同完成，要么不执行。这种现象也称为**原子性**

(atomicity),在事务中广泛应用。就像下面介绍的那样,除了事务以外,其他应用程序也可以利用分布式提交协议,比如具有容错性的应用程序。

最后一个例子是关于分布式锁定协议的,这种协议可以保护资源免受分布于多台机器上的一组进程的并发访问。我们将会在第 6 章中介绍这种协议。同样,该协议也可以用于实现通用中间件服务,但同时又是高度独立于任何特定应用程序的示例。

中间件通信协议支持高层通信服务。例如,在接下来的两小节我们将要讨论允许进程以高度透明的方式调用远程机器上的过程或者调用远程机器上的对象的协议。同样,也有用于对实时数据传输进行设定并使其保持同步的协议,这种实时数据传输是多媒体之类的应用程序所需要的。最后,某些中间件系统还提供可靠的多播服务,这种服务的规模可以达到遍及广域网上数以千计的接收方。

某些中间件通信协议也可以归属于传输层,而把它们归属于更高层还有一些特殊的原因。比如,可扩展性得到保证的可靠的多播服务只在考虑应用程序需求的情况下才能实现。因此,中间件系统可以提供不同的(可调整的)协议,而每种协议则通过不同的传输协议来实现,但只提供一个接口。

采用这种方案进行分层将形成一种改进的通信参考模型,如图 4.3 所示。与 OSI 模型相比,会话层和表示层由一个单一的中间件层代替,该层中包含有与应用程序无关的协议。这些协议不属于我们刚才讨论过的低层。原来的传输服务可以作为一个中间件服务来提供,无需改动。这种方案与在传输层提供的 UDP 有异曲同工之处。同样,中间件通信服务中可以包含消息传递服务,这与由传输层提供的相似。

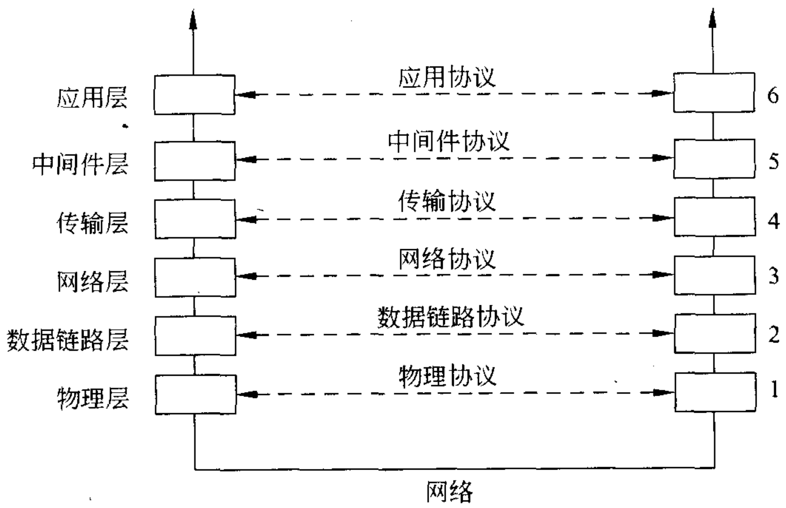


图 4.3 调整后的网络通信参考模型

在本章的余下部分中,我们将集中讨论 4 种高级的中间件通信服务:远程过程调用、远程对象调用、消息队列服务以及通过流和多播为连续媒体通信提供的支持。在介绍这些内容之前,下面先来介绍区分(中间件)通信的其他常见标准。

4.1.2 通信类型

为了理解中间件为应用程序提供的各种通信方式,可以把中间件视作为客户-服务器计

算的另一种服务，如图4.4所示。以电子邮件系统为例。原理上，邮件传送系统的核心可以看作是一种中间件通信服务。每个主机运行一个用户代理，允许用户编写、发送和接收E-mail。发送用户代理把邮件传递给邮件传送系统，希望它最终把邮件传送给目标接受者。同样，在接收端，连接到邮件传送系统的用户代理将检查是否有邮件到来。如果有，就把邮件转移到用户代理，这样就可以被用户显示和读取。

电子邮件系统是一个其通信为持久的典型示例。对于**持久通信**(persistent communication)，提交传输的消息一直由通信中间件存储，直到该消息被传送给接收方为止。在这种情况下，中间件将在一个或多个存储设备中存储该消息，如图4.4所示。因此，对发送应用程序来说，在消息提交后，就没有必要持续运行了。同样，消息提交后，接收应用程序也没有必要持续运行。

相反，对于**瞬时通信**(transient communication)，通信系统只有在发送和接收应用程序正在运行时才存储消息。更准确地说，根据图4.4所示，由于传输中断或因为接收方当前不在活动状态，中间件就不能传送消息了，而是把它丢弃掉。通常，所有传输层通信服务都只提供瞬时通信。在这种情况下，通信系统由传统的存储和转发路由器。如果某个路由器无法把消息传送到下一个路由器或目标主机，那么它将丢弃该消息。

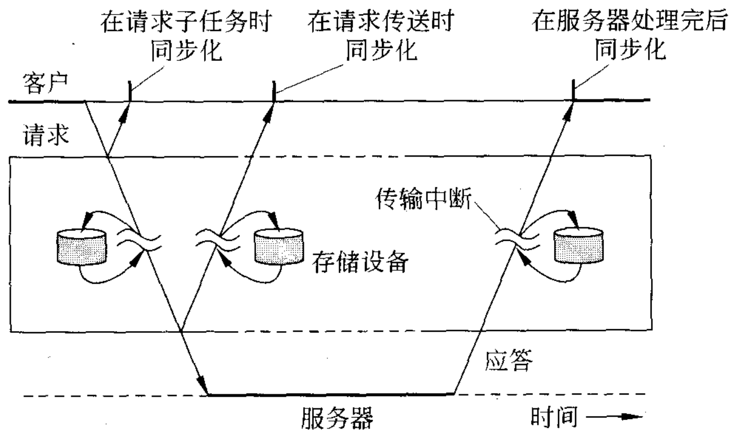


图4.4 在应用层通信中把中间件视作为一种中间(分布式)服务

通信除了可以是持久或瞬时的之外，还可以是同步或异步的。**异步通信**(asynchronous communication)的特征是，发送方在提交要传输的消息后立即往下进行。这意味着消息在提交后立即由中间件(临时)存储起来。对于**同步通信**(synchronous communication)，发送方将被阻塞，直到知道其请求被接受以后。同步通信发生时基本有三点。首先，发送方可能被阻塞，直到中间件通知它请求传输已完成。第二，发送方为同步化，直到其请求被传送给目标接收方。第三，通过让发送方一直等到其请求被完全处理，也就是说，直到接收方返回了一个响应，就可以实现同步化。

在实际应用中，会出现持久通信和同步通信的各种组合。最为常见的是持久与请求提交时同步化的组合，对很多消息队列系统来说，这是一种常用的方案，本章后面将会讨论。同样，瞬时通信与在请求完全处理后的同步化的组合使用也很广泛。这种方案与远程过程调用相符，随后将讨论它。

除持久性和同步化之外，我们还要区分不连续和流通信。到目前为止的所有示例都属于不连续通信：各方以消息进行通信，每个消息组成一个完整的信息单元。流通信则可以

一个接一个地发送多个消息，这些消息按它们的发送顺序是相互有关联的，或者是一种临时关系。后面将进一步介绍流通信。

4.2 远程过程调用

许多分布式系统是基于进程间的显式消息交换的，然而，消息的发送(send)和接收(receive)过程无法隐藏通信的存在，而通信的隐藏对于在分布式系统中实现访问透明性是极为重要的。这个问题由来已久，但是人们在很长一段时间内都没有找到合适的解决办法，直到(Birrell 和 Nelson 1984)在一篇论文中引入了一套与传统方法截然不同的通信处理手段。虽然其中蕴含的思想很简单(一旦经过思考以后就很容易理解)，但其中的含义非常复杂精妙。在本节中，我们将详细分析这个概念，包括它的实现、长处以及弱点。

简要地说，Birrell 和 Nelson 认为应该允许程序调用位于其他机器上的进程。当机器 A 上的进程调用机器 B 上的进程时，A 上的调用进程被挂起，而 B 上的被调用进程开始执行。调用方可以通过使用参数将信息传送给被调用方，然后可以通过传回的结果得到信息。编程人员看不到任何消息传递过程。这种方法称为**远程过程调用**(remote procedure call，RPC)。

虽然基本思想听起来很简单也很不错，但是还存在很多的具体问题。首先，调用过程和被调用过程运行在不同机器上，它们在不同的地址空间内执行，这就带来了复杂性。其次，需要进行参数和结果传递，这种传递可能是复杂的，特别是在机器间存在差异的情况下更是这样。最后，双方所在的机器都可能发生崩溃，任何一种潜在的故障都可能引发不同的问题。尽管存在这么多问题，但是其中的大多数是能够解决的。RPC 作为一种广泛使用的技术，已成为许多分布式系统的基础。

4.2.1 基本的 RPC 操作

我们先讨论常规的过程调用，然后解释如何把调用本身划分为分别在不同机器上执行的客户部分和服务器部分。

4.2.1.1 常规过程调用

为了理解 RPC 的工作机制，首先必须彻底理解常规(也就是单机)过程调用的工作机制。考虑下面的 C 语言调用：

```
count = read (fd, buf, nbytes);
```

其中，fd 为一个整型数，表示一个文件。buf 为一个字符数组，用于存储读入的数据。nbytes 为另一个整型数，用来记录实际读入的字节数。如果该调用位于主程序中，那么在调用之前堆栈的状态将如图 4.5(a)所示。为了进行调用，调用方首先把参数反序压入堆栈，即最后一个参数最先压入，如图 4.5(b)所示。(C 编译器将参数按相反顺序压入堆栈的原因与函数 printf 有关。因为只有这样做，才能保证 printf 能找到它的第一个参数，即格式

字符串。)在 read 操作运行完毕后，它将返回值放在某个寄存器中，移出返回地址，并将控制权交回调用方。调用方随后将参数从堆栈移出，使堆栈还原到初始的状态。

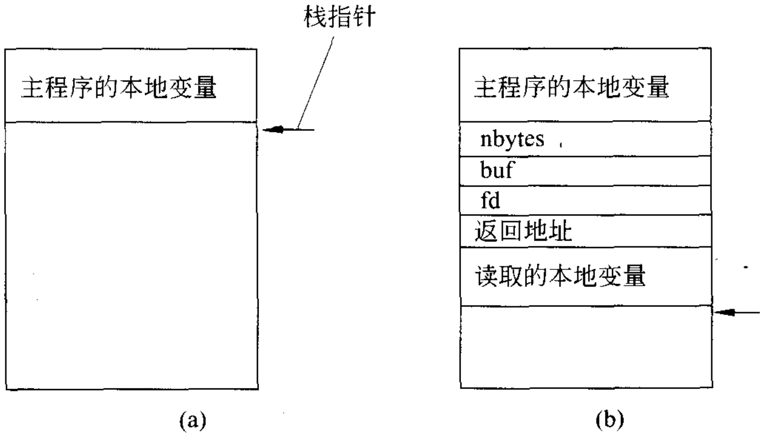


图 4.5 过程调用中的参数传递

(a) 本地过程调用中的参数传递：调用 read 之前的堆栈状态；(b) 被调用过程执行时的堆栈状态

有一些问题需要注意。其中一个问题是，在 C 语言中，可以采用**传值调用**(call-by-value)和**引用调用**(call-by-reference)来传递参数。数值参数，如上面例子中的 fd 和 nbytes，只是简单地复制到栈中，如图 4.5(b)所示。对于被调用过程来说，数值参数仅仅相当于已初始化的局部变量，被调用过程可以对这种参数进行修改，但是所作的改动并不会影响到调用端该参数原来的值。

C 语言中的引用参数实际上是指向某个变量的指针(也就是该变量所在的地址)，而不是该变量的值。在上例对 read 的调用中，第二个参数就是一个引用参数，因为 C 语言中的数组总是通过引用传递的。实际上，压入堆栈的是字符数组的地址。如果被调用过程使用该参数来将某些内容存储到字符数组中去的话，它确实修改了位于调用者过程中的数组。对于 RPC 来说，传值调用与引用调用的区别显得尤为重要，我们将在下文中看到这一点。

还存在另一种在 C 语言中未使用的参数传递机制，称为**复制-还原调用**(call-by-copy/restore)。这种调用的过程是：首先由调用方将变量复制到堆栈中，这一步与传值调用相同；随后在调用完毕之后将堆栈中的变量复制回去，覆盖掉调用方中该变量原先的值。多数情况下，这种调用的结果与引用调用相同，但是某些情况下这种调用会有不同的语义，比如说如果同一个参数在参数列表中出现了多次就会造成不同。许多语言中没有使用复制-还原调用机制。

采用哪种参数传递机制的决策通常由语言的设计者做出。参数传递机制是语言的固定特性之一，有些时候它还依赖于要传递的数据类型。在 C 语言中，整型数以及其他的标量类型总是通过值传递的，而数组总是通过引用传递的，正如我们刚才看到的那样。某些 Ada 编译器对于**入出**(in out)参数采用复制-还原调用机制，而其他编译器则采用引用调用机制。由于语言本身的规定同时允许这两种参数传递机制，这就使得在语义上显得有些模糊。

4.2.1.2 客户与服务器存根程序

RPC 背后隐含的思想是尽量使远程过程调用具有与本地调用相同的形式。也就是说，希望 RPC 是透明的，即调用过程不应该发现被调用的过程实际在另一台机器上执行，反过

来也一样。假定程序需要从某个文件读取数据，程序员在代码中执行 read 调用来取得数据。在传统(单处理器)系统中，read 例程由链接器从库中提取出来，然后链接器再将它插入目标程序中。read 过程是一个短过程，一般通过执行一个等效的 read 系统调用来实现。也就是说，read 过程是一种位于用户代码与本地操作系统之间的接口。

虽然 read 中执行了系统调用，但是它本身依然是通过将参数压入堆栈的常规方式调用的，如图 4.5(b)所示。这样，程序员就不会知道 read 究竟干了些什么。

RPC 是通过类似的途径获得透明性的。当 read 实际上是一个远程过程时(比如在文件服务器所在机器上运行的过程)，库中就放入 read 的另一种版本，称为**客户存根**(client stub)。这种版本的 read 过程同样遵循图 4.5(b)中的调用次序，这一点与原来的 read 过程相同，另外一个相同点是其中也执行了本地操作系统调用。唯一的不同点就是它并不要求操作系统提供数据，而是将参数打包成一个消息，然后请求将此消息发送到服务器，如图 4.6 所示。在对 send 的调用之后，客户存根调用 receive 过程，随即阻塞自己，直到收到响应消息。

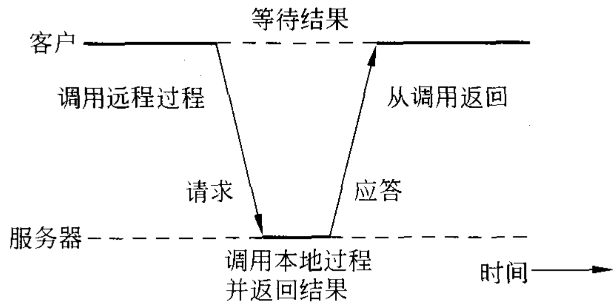


图 4.6 客户与服务器之间的 RPC 的原理

当消息送达服务器时，服务器上的操作系统将它传递给**服务器存根**(server stub)。服务器存根是客户存根在服务器端的等价物，它也是一段代码，用来将通过网络输入的请求转换为本地过程调用。服务器存根一般先调用 receive，然后被阻塞，等待消息输入。收到消息后，服务器存根将参数由消息中提取出来，然后以常规方式调用服务器上的相应过程(如图 4.6 所示)。从服务器的角度来看，过程就好像是由客户直接调用一样：参数和返回地址都位于堆栈中，一切都很正常。服务器执行所要求的操作，随后将得到的结果以常规的方式返回给调用方。以 read 为例，服务器将用数据填充 read 中第二个参数指向的缓冲区，该缓冲区是属于服务器存根内部的。

调用完毕之后，服务器存根要将控制权交回给客户发出调用的过程，它将结果(缓冲区)打包成消息，随后调用 send 将结果返回给客户。此后，服务器存根一般会再次调用 receive，等待下一个输入的请求。

客户机器接收到消息以后，客户操作系统会发现该消息属于某个客户进程(实际上该进程是客户存根，但是操作系统无法对二者做出区分)。操作系统将消息复制到相应的缓冲区中，随后解除对该客户进程的阻塞。客户存根检查该消息，将结果提取出来并复制给调用者，然后以通常的方式返回。当调用者在 read 调用进行完毕后重新获得控制权时，它所知道的唯一的事就是已经得到了所要的数据。它并不知道操作不是通过本地操作系统进行，而是在远程完成的。

整个方法的精彩之处在于客户方可以简单地忽略不关心的内容。客户所涉及的操作只是通过执行普通的（本地）过程调用来访问远程服务，它并不需要直接调用 send 和 receive。消息传递的所有细节都隐藏在双方的库过程中，就像传统库中隐藏了执行实际系统调用的细节一样。

总的说来，远程过程调用包含下列步骤：

(1) 客户过程以正常的方式调用客户存根；

(2) 客户存根生成一个消息，然后调用本地操作系统；

(3) 客户端操作系统将消息发送给远程操作系统；

(4) 远程操作系统将消息交给服务器存根；

(5) 服务器存根将参数提取出来，然后调用服务器；

(6) 服务器执行要求的操作，操作完成后将结果返回给服务器存根；

(7) 服务器存根将结果打包成一个消息，然后调用本地操作系统；

(8) 服务器操作系统将含有结果的消息发送回客户端操作系统；

(9) 客户端操作系统将消息交给客户存根；

(10) 客户存根将结果从消息中提取出来，返回给调用它的客户过程。

所有这些步骤总的效果是，将客户过程对客户存根发出的本地调用转换成对服务器过程的本地调用，而客户和服务器都不会意识到有中间步骤的存在。

4.2.2 参数传递

客户存根的功能是将得到的参数打包成消息，然后将消息发送给服务器存根。虽然这听起来很清楚明了，但是实际上并不那么简单。在本节中，我们将讨论 RPC 系统中涉及的一些参数传递方面的问题。

4.2.2.1 传递值参数

把参数打包进消息中的过程称为**参数列集**(parameter marshaling)。作为一个非常简单的例子，我们来考虑一下远程过程 add(i, j)。该过程有两个整型参数 i 和 j，在结果中返回 i 和 j 的算术和（实际上，基于开销方面的考虑，一般不会对这种简单过程进行远程调用，这里只是作为一个简单例子）。在客户进程中对 add 的调用如图 4.7 中左边部分所示。客户存根将 add 的两个参数放入消息中，同时还在消息中添加要调用的过程的名称或者编号，这是为了在服务器支持多个不同调用的情况下指明要求执行的是哪一个调用。

消息送达服务器之后，服务器存根对该消息进行分析，以判明需要调用哪一个过程，随后执行相应的调用。如果服务器还支持其他的远程过程，服务器存根中就需要使用 switch 语句，以根据消息中首字段的内容来选取需要调用的过程。服务器存根对服务器的实际调用与原来客户进程发出的调用十分相似，不同的只是前者的参数是依据输入的消息来完成初始化的变量。

服务器运行完毕后，服务器存根重新取回控制权，并将服务器得到的结果打包成消息送回给客户存根，客户存根随即将结果从消息中提取出来，把结果值返回给客户过程。

只要客户机器与服务器机器没有差异，并且所有的参数和结果都是标量类型，比如整

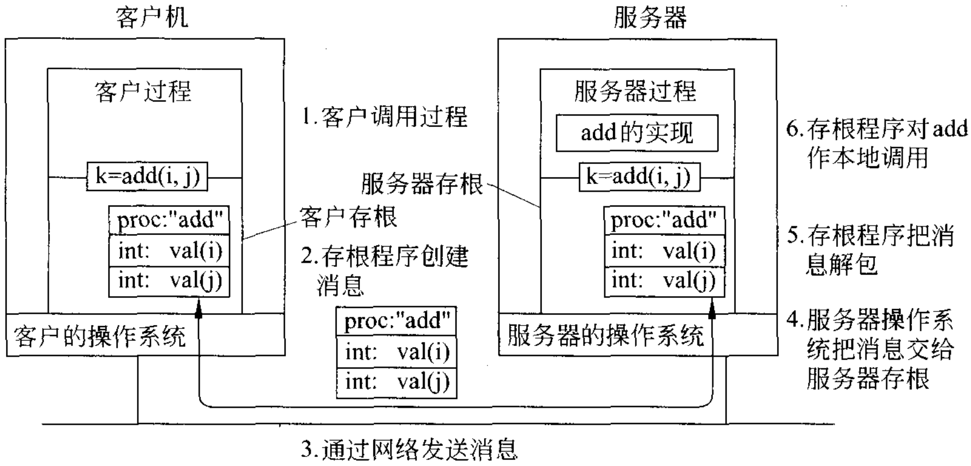


图 4.7 通过 RPC 进行远程计算的步骤

型、字符型和布尔型，以上模型就可以正常工作。然而，在大型分布式系统中，存在多种不同类型的机器是司空见惯的。不同的机器对于数字、字符以及其他种类的数据项的表示方式常有不同。比如，IBM 大型机使用 EBCDIC 字符码，而 IBM 个人电脑使用 ASCII 字符码。因此要以图 4.8 中所示的简单方式从 IBM PC 客户向 IBM 大型机传递字符型参数是不可能的，服务器将会错误地解释该参数中的字符。

类似的问题在整型数表示方式(是以 1 的补码还是以 2 的补码表示)以及浮点数表示方式方面也存在。另外，还存在另一个更为棘手的问题，就是字节排列次序问题。包括 Intel 奔腾在内的某些处理器是从右向左排列字节的，而 Sun SPARC 等处理器则相反。Intel 采用的格式称为**小端**(little endian)格式，而 SPARC 采用的格式称为**大端**(big endian)格式，这些格式的称呼来自于《格利佛游记》，书中描述的政治家决定开战的理由只是一方从大端来敲鸡蛋，而另一方从小端敲鸡蛋(Cohen 1981)。举例来说，考虑一个使用两个参数的过程，参数分别是一个整型数和一个包含 4 个字符的字符串。每个参数都需要占用一个 32 位的字。图 4.8(a)显示了基于 Intel 奔腾处理器的客户存根生成的消息中的参数部分的内容：第一个字包含整型参数，其值为 5，而第二个字中包含字符串“JILL”。

由于消息是逐字节(实际上是逐位)通过网络进行传输的，第一个发送的字节将第一个送达。在图 4.8(b)中显示了由 SPARC 接收到的消息内容。SPARC 将字节从左至右排列，0 字节放在最左边(高阶字节)；而不是像所有 Intel 处理器那样将字节从右至左排列，0 字节放在最右边(低阶字节)。当 SPARC 机器上的服务器存根读出地址分别是 0 和 4 的参数值时，它将得到一个值为 83 886 080(5×2^{24})的整型数，以及一个字符串“JILL”。

简单地将接收到的每个字中的字节顺序逆转过来，如图 4.8(c)所示，虽然很容易想到，但却是错误的。如果这样做，整型数虽然变回了 5，但是字符串却变成了“LLIJ”。关键问题

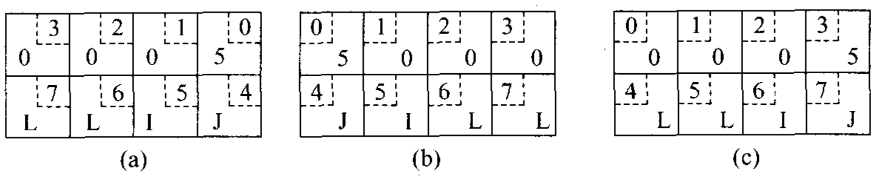


图 4.8 原始的消息和逆转后的消息(方框中的小数字代表每个字节的地址)

(a) 奔腾处理器上原始的消息；(b) SPARC 接收到的消息；(c) 进行逆转以后的消息

是虽然机器间字节排序的差异把整型数弄错了，但是字符串并没有受到影响。如果没有用来识别字符串和整数的附加信息，就无法对该损害进行修复。

4.2.2.2 传递引用参数

现在我们面临着一个难题：指针（广义来说还包括引用）如何传递？答案是：即使这种传递是可能的，也必定是极为困难的。要知道，指针只在使用该指针的进程所在的地址空间内才有意义。回头看看早先那个关于 read 的例子，如果第二个参数（缓冲区地址）在客户端为 1000，仅仅把 1000 这个数值传给服务器是没有用处的。服务器上的地址 1000 可能正好位于程序文本的中间。

解决这个问题的一种方法是完全禁止使用指针和引用，但是由于指针和引用是非常重要的，因此这种解决方法极为不可取。事实上，在上面那个 read 调用的例子中，客户存根知道作为第二个参数的指针指向的是一个字符数组。假定此时它还知道该数组的大小，下面的解决策略显而易见：将数组复制到消息中，发送给服务器。服务器存根随后使用指向数组的指针来调用服务器程序，即使该指针拥有与 read 的第二个参数不同的数字值。服务器使用指针所作的改动（例如，在其中存储数据）直接影响了属于服务器存根的缓冲区。当服务器完成操作之后，服务器存根将原先的消息送回给客户存根，客户存根再将它复制回客户进程。这种方法实际上是用复制-还原调用来代替引用调用，虽然不完全相同，但是一般来说是行之有效的。

以下的优化策略可以使这种机制的效率提高一倍。如果存根知道缓冲区对服务器来说是输入参数还是输出参数，就可以省略其中的一个复制步骤。如果数组对服务器来说是输入参数（比如在 write 调用中就是这样），就不需要将数组复制回客户存根。如果数组是输出参数，那么客户存根最初发送的消息中就不用加进数组。

最后要指出的一点是：要充分认识到，虽然目前我们提出了处理指向简单数组和结构的指针的办法，但是我们还是无法处理一般意义上的指针，即指向任意数据结构，例如复杂图形的指针。某些系统试图将指针直接传递给服务器存根，然后在该服务器过程中生成特殊代码以使用这种指针。例如，服务器可以向客户发回一个请求，要求提供引用的数据。

4.2.2.3 参数说明和存根生成

从根据前面的解释可以清楚地看到，要隐藏对远程过程调用，必须使调用者和被调用者就互相交换的消息格式达成一致，并且在进行诸如传递复杂数据结构之类的操作时遵循相同的步骤。换句话说，RPC 的双方必须遵循相同的协议。

作为一个简单的例子，看一下图 4.9(a)中的过程。该过程包含 3 个参数，分别为字符、浮点数和含有 5 个整数的数组。假定一个字的长度为 4 个字节，RPC 协议可以规定：把字符放在一个字最右边的字节内传输（其余 3 个字节为空）；浮点数占一个字；而数组由一组字组成，字的个数与数组长度相同，前面加上一个说明数组长度的字，如图 4.9(b)所示。在给出这些规则之后，名为 foobar 的客户存根就会知道它必须使用图 4.9(b)中规定的格式，而服务器存根也会知道输入的消息将使用图 4.9(b)中规定的格式。

对消息格式的规定只是 RPC 协议内容中的一个方面。还需要让客户和服务器就诸如整型数、字符、布尔量等简单数据结构的表示方法达成一致。例如，协议可以规定整型数用

```
foobar(char x; float y; int z[5])
{
  ...
}
```

(a)

foobar的本地变量
x
y
5
z[0]
z[1]
z[2]
z[3]
z[4]

(b)

图 4.9 过程及其对应的消息

(a) 一个过程；(b) 与该过程相对应的消息

2 的补码形式表示，字符用 16 位 Unicode 编码表示，浮点数使用 IEEE 754 号标准规定的格式，并且所有数据都使用小端格式存储。有了这些信息，就可以对消息进行明确的解释，而不会产生歧义。

在编码规则已经详尽到每一位之后，唯一剩下的事情就是让调用者和被调用者就消息的实际交换达成一致。例如，可以决定采用诸如 TCP/IP 之类的面向连接的传输服务，也可以采用不可靠的数据报服务。如果是后者的情况，必须由客户和服务器自己实现一套错误控制方案，并将该方案作为 RPC 协议的一部分。在实践中还存在许多其他方案。

在完整地定义了 RPC 协议之后，需要实现客户存根和服务器存根。幸运的是，相同协议所使用的存根在用于不同的过程时，不同点仅仅在于面向应用程序的接口。接口由一组由服务器实现的可供客户调用的过程组成，可以用编写客户或者服务器的同一种编程语言来编写（虽然严格说来这并不必要）。为了进一步简化，接口通常使用**接口定义语言**(interface definition language，IDL)来说明。用 IDL 说明的接口可以与适当的编译时接口或者运行时接口一起编译到客户存根过程和服务器存根中。

在实践中发现，使用接口定义语言可以显著简化基于 RPC 的客户-服务器应用程序。由于生成客户存根和服务器存根很容易，所有基于 RPC 的中间件系统都提供 IDL，以对应用程序开发提供支持。在某些情况下，更是要强制使用 IDL，这一点将在以后的章节中介绍。

4.2.3 异步 RPC

与在常规过程调用中的情形一样，当客户调用远程过程时，客户将会阻塞，直到有应答返回为止。在没有结果要返回的情况下这种严格的请求-应答方式是不必要的，它只会导致客户过程向远程过程发出调用请求之后处于阻塞状态，从而无法进行本来能够进行的其他有用的工作。不需要等待应答的例子有：从某个账户向另一个账户进行转账操作、向数据库添加条目、启动远程服务以及执行批处理操作等。

为了支持以上这些状况，RPC 系统可以提供称为**异步 RPC**(asynchronous RPC)的功能。利用该功能，客户可以在发出 RPC 请求后立即继续执行。在异步 RPC 中，服务器在接收到 RPC 请求后立即向客户送回应答，之后再调用客户请求的过程。应答的作用是向客户

确认服务器已准备开始处理该 RPC 请求。客户接收到服务器的确认消息之后，将不会阻塞，而是继续向下执行。图 4.10(b)中显示了异步 RPC 中客户和服务器的交互过程。作为对比，图 4.10(a)中显示了常规 RPC 的请求-应答过程。

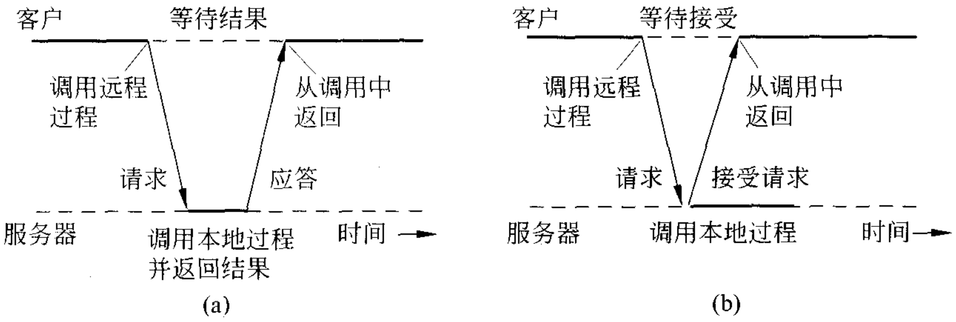


图 4.10 RPC 中客户-服务器间的交互过程
(a) 传统 RPC 中客户-服务器间的交互过程；(b) 使用异步 RPC 时的交互过程

如果在应答返回时客户还未做好接收的准备，从而没有做出任何动作，在这种情况下异步 RPC 也很有用。例如，客户可能想预取一组主机的网络地址以便随后与它们进行联络。当命名服务正在搜集这些地址的时候，客户可能想做其他事情。在这种情况下，通过两步异步 RPC 来进行客户与服务器间的通信是很有意义的，如图 4.11 所示。客户首先对服务器进行调用，把要查询的主机名清单交给服务器，并且客户在接收到服务器关于已收到主机名清单的确认之后继续执行其他的程序。第二步调用是由服务器发出的，它对客户进行调用，将查询到的地址清单交给客户。两步异步 RPC 结合起来也称为**延迟的同步 RPC**(deferred synchronous RPC)。

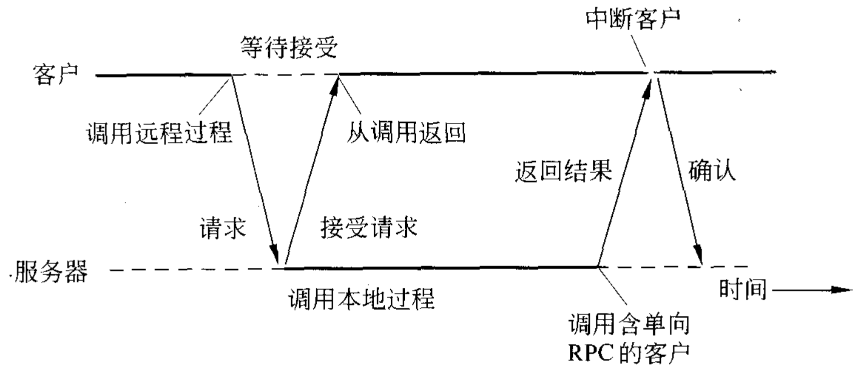


图 4.11 客户和服务器通过两个异步 RPC 进行交互

需要指出的是，还存在一些异步 RPC 的变异形式。在这些异步 RPC 中，客户向服务器发送请求之后立即继续执行其他的程序。也就是说，客户不等待服务器返回接受请求的确认。我们称这种 RPC 为单向 RPC(one-way RPC)。它的问题在于，如果无法确保可靠性，客户就无法确定它发出的请求是否将得到处理。我们将在第 7 章中继续讨论这些问题。

4.2.4 示例：DCE RPC

远程过程调用已广泛用作大部分中间件和分布式系统的基础。在本节中，我们重点讨

论一个具体的 RPC 系统：**分布式计算环境**(distributed computing environment,DCE)。DCE 是由**开放软件基金会**(Open Software Foundation,OSF)开发的,该组织现在称为**开放式开发组**(Open Group)。DCE RPC 的流行程度比不上其他某些 RPC 系统,特别是 Sun RPC。但是,DCE RPC 是 RPC 系统的一个典型代表,其规范得到了 Microsoft 的分布式计算基本系统的采用。另外,DCE RPC 可以很好地说明 RPC 系统与分布式对象间的联系,这点在后面一节中可以看出。我们首先简要介绍 DCE,随后仔细考虑 DCE RPC 的主要工作机制。

4.2.4.1 DCE 概述

DCE 是设计用于作为在现有的(网络)操作系统与分布式应用程序之间的中间抽象层来执行的。从这个意义上来说,DCE 是真正的中间件系统。DCE 最初是为 UNIX 设计的,但是现在已经移植到了所有主要操作系统上,包括 VMS、Windows NT 以及桌面操作系统在内。它的主要思想是,使用者可以在现有的一组机器上安装 DCE 软件,然后就可以运行分布式应用程序,而不会对现有的(非分布式的)应用程序造成影响。虽然大部分 DCE 包在用户空间运行,但是在采用某些特定配置的情况下,必须在内核中添加一部分内容(分布式文件系统的一部分)。开放式开发组自己只是出售源代码,由销售商将源代码集成进他们的系统中去。

所有 DCE 的底层编程模型都是客户-服务器模型,该模型在第 3 章中已经详细讨论过了。用户进程就像客户那样对服务器进程提供的远程服务进行访问。其中的某些服务属于 DCE 的一部分,而其他服务属于应用程序,由应用程序开发人员编写。客户和服务器间的所有通信都通过 RPC 进行。

DCE 本身的一部分是由多个服务构成的。**分布式文件服务**(distributed file service)是一种全球规模的文件系统,可以提供一种透明的访问方式,允许使用相同手段访问系统中任意文件。分布式文件服务既可以建立在主机的本地文件系统之上,也可以用来代替本地文件系统。**目录服务**(directory service)是用来跟踪系统中所有资源的位置的,这些资源包括机器、打印机、服务器、数据等,它们在地域上可以分布在全世界范围内。目录服务允许一个进程在不了解某个资源所在位置的情况下(除非进程关心这件事)请求该资源。**安全服务**(security service)允许对所有类型的资源进行保护,这样就可以将访问权提供给已授权的用户。最后,**分布式时间服务**(distributed time service)可以使位于不同机器上的时钟保持全局同步。正如我们将在后续章节中介绍的那样,全局时钟的思想使得在分布式系统中确保一致性变得更加容易。

4.2.4.2 DCE RPC 的目标

DCE RPC 系统的目标是比较传统的。首先同时也是最重要的是,RPC 系统使客户可以通过简单的本地过程调用来访问远程服务。这种接口使得客户程序(例如应用程序)能够以一种简单的、为多数程序员所熟悉的方式来编写,还使得大量现有的代码不需要进行改动(或者只需要极少的改动)就能够在分布式环境下运行。

RPC 系统负责向客户隐藏所有细节,并且也在某种程度上对服务器进行细节隐藏。RPC 系统首先自动找到恰当的服务器,随后在客户软件和服务器软件之间建立通信(这个

过程一般称为**绑定**(binding))。它还能处理双向数据传输，在必要的情况下把消息分割成片段，然后再将片段重组还原成消息(比如说，在参数之一是大数组的情况下)。最后，RPC系统可以自动完成客户和服务器间的数据类型转换，这样，即使客户和服务器运行在体系结构不同而且字节排序方式不同的机器上也没有问题。

由于RPC系统具有隐藏细节的能力，所以客户和服务器是彼此高度独立的。可以用Java编写客户程序而用C编写服务器程序，反之亦然。客户和服务器可以运行于不同的硬件平台上，并且可使用不同的操作系统。它还支持非常多的网络协议和数据表示方式，而不需要客户和服务器进行任何干预。

4.2.4.3 编写客户程序和服务器程序

DCE PRC系统由许多组件组成，其中包括语言组件、库、守护程序以及工具程序等。配合使用这些组件就可以编写客户程序和服务器程序。本节将对各个部分以及它们相互配合的方式进行描述。图4.12总结了编写以及使用一个RPC客户程序和服务器程序的完整过程。

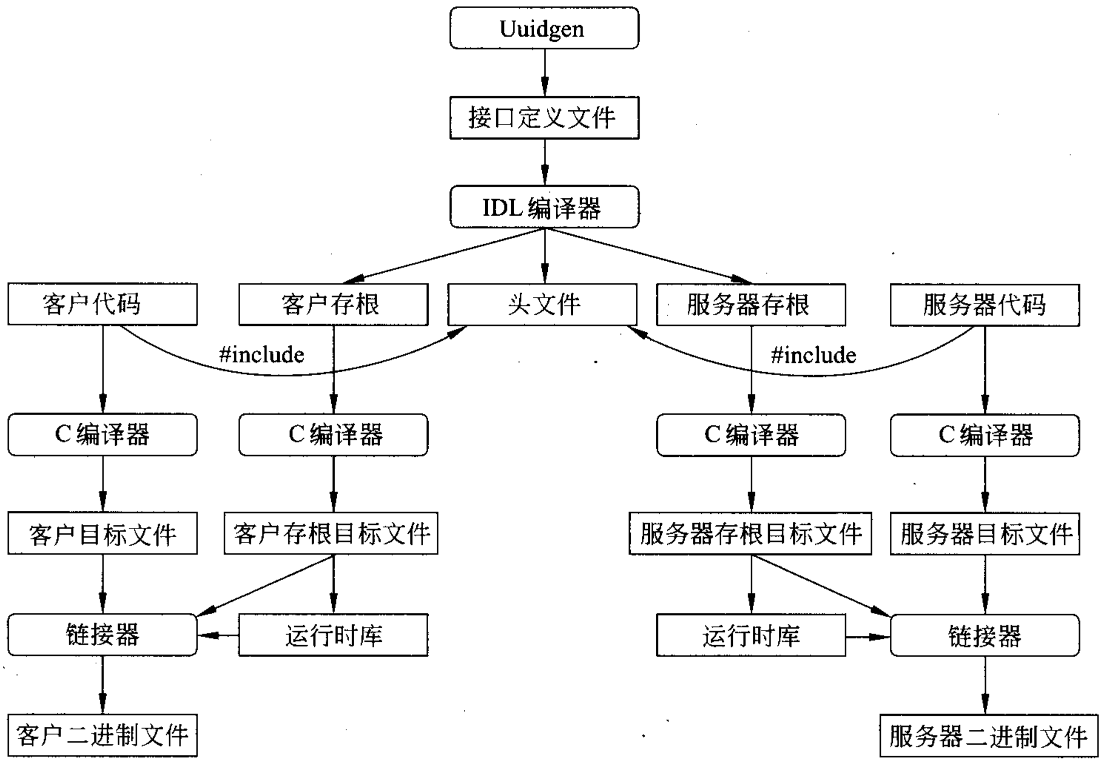


图4.12 用DCE RPC编写客户程序和服务器程序的步骤

在客户-服务器系统中，将各个部分结合到一起的是使用**接口定义语言**(Interface Definition Language，IDL)描述的接口定义。这种语言允许以极其类似于ANSI C语言中的函数原型的形式来对过程进行说明。IDL文件也可以包含类型定义、常量声明以及正确进行参数编组和结果解编时所必需的其他信息。理想情况下，接口定义中还应该含有过程所完成的工作的形式化定义，但是这样的定义在目前的技术条件下还是无法做到的。因此接口定义只是规定了调用的格式，而不包括调用的语义。编写者顶多能够添加少量注释，以描述该过程要完成的工作。

每个IDL文件中关键的部分是指定接口的全局唯一标识符。在第一个RPC消息中，客户把该标识符发送给服务器，由服务器检验该标识符是否正确。如果当某个客户无意中试图绑定到错误的服务器上或者是正确服务器的一个过时版本上时，服务器将会检测到该错误，绑定将不会建立。

接口定义和唯一标识符在DCE中是紧密联系在一起的。就像图4.12中显示的那样，编写服务器或客户应用程序的第一步常常是调用uuidgen程序，要求该程序生成一个原型IDL文件，其中包含有一个接口标识符，该标识符保证不会在由uuidgen生成的任何其他接口中再次使用。在该标识符的编码过程中加进了它所在的位置和创建时间，从而确保了它的唯一性。该标识符是一个128位的二进制数，在IDL文件中表示为一个十六进制的ASCII字符串。

下一步是编辑IDL文件，填入远程过程名以及它们使用的参数。值得注意的是，RPC并不是完全透明的(例如，客户和服务器无法共享全局变量)，而IDL的定义规则已经保证不可能表示那些不支持的结构。

IDL文件编写完毕之后，可以调用IDL编译器来编译它。IDL编译器的输出包括下列3个文件：

(1) 头文件(例如，C语言形式的interface.h)；

(2) 客户存根；

(3) 服务器存根。

头文件中包含唯一标识符、类型定义、常量定义以及函数原型。应该在客户和服务器代码中使用#include语句将该头文件包含进去。客户存根包含供客户程序调用的实际过程，这些过程负责参数搜集，并且将搜集到的参数打包进准备送出的消息中，随后调用运行时系统来发送该消息。客户存根还负责将结果值从应答中提取出来，返回给客户。服务器存根包含有一些过程，在输入的消息到达时，由服务器所在机器上的运行时系统调用这些过程，这些过程随后再调用服务器过程来完成实际操作。

接下来是由应用程序编写人员编写客户和服务器代码。要对服务器、客户以及双方的存根进行编译。然后将得到的客户代码和客户存根的目标文件与运行时库连接起来，从而得到客户程序的二进制可执行形式。与之相仿，将服务器代码和服务器存根的代码进行编译和连接，可以得到服务器程序的二进制可执行形式。在运行时，把客户程序和服务器程序都启动起来，这样应用程序就可以正确执行了。

4.2.4.4 将客户绑定到服务器

为了让客户能够调用服务器，必须先对服务器进行注册，然后服务器做好接收输入调用的准备。服务器进行了注册之后，客户就能够对服务器进行实际定位，然后绑定到服务器。服务器定位分两步进行：

(1) 定位服务器所在的机器；

(2) 定位该机器上的服务器(也就是正确的进程)。

第2步稍微有些复杂，但基本上可以归结为这样一个问题：为了与服务器通信，客户需要知道服务器所在机器上的一个**端点**(endpoint)，以便向服务器所在机器发送消息。服务器的操作系统使用端点(通常称为**端口**(port))来判别输入的消息要调用哪一个进程。在

DCE 中，每一台运行服务器的机器上都由 **DCE 守护程序**(DCE daemon)维护一张(服务器，端点)对列表。服务器在可以接收输入请求之前，必须先请求操作系统分配一个端点，随后通过 DCE 守护程序注册该端点。DCE 守护程序在端点列表中记录下有关信息(包括服务器采用何种协议)供以后使用。

服务器同时还需要通过目录服务进行注册，方法是给出服务器所在机器的网络地址以及在查询该服务器时使用的名字。把客户绑定到服务器的过程如图 4.13 所示。

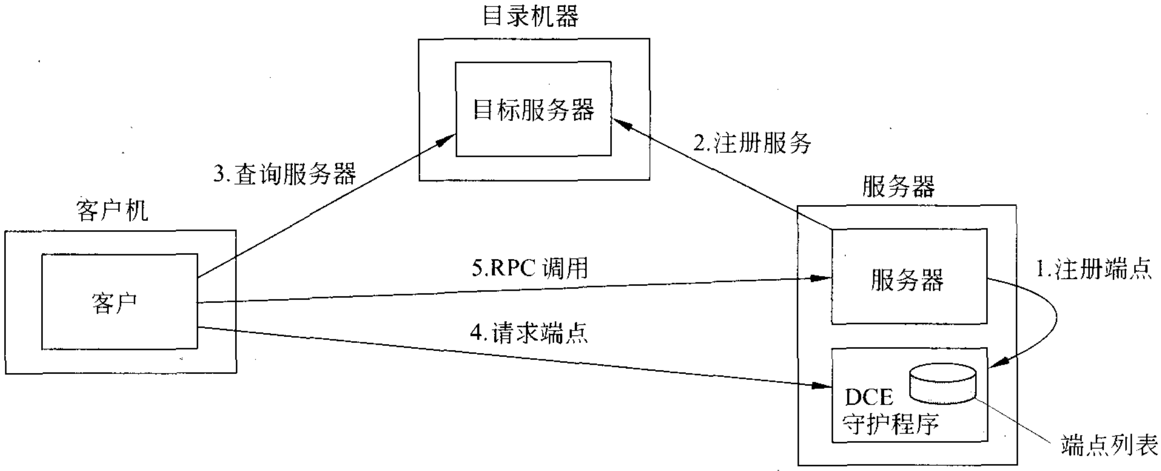


图 4.13 DCE 中的客户-服务器绑定过程

假定客户希望绑定到一个视频服务器上，该服务器的本地名是/local/multimedia/video/movies。客户把该名字传递给目录服务器，目录服务器返回运行视频服务器的机器的网络地址。客户随后向该机器上的 DCE 守护程序(它的端点是公开的)发出请求，要求它在端点列表中查询视频服务器使用的端点。利用所获得的信息，就可以执行 RPC 了。在随后的执行 RPC 的过程中，不再需要进行查询。如果有必要，DCE 还允许客户为了查找一个合适的服务器执行较为复杂的搜索。另外，还可以选择执行安全的 RPC。

4.2.4.5 执行 RPC

实际的 RPC 过程是以常规方式透明地执行的。客户存根对参数进行编组，然后把它们交给运行时库，以使用在绑定时选定的协议来传输这些参数。当消息送达服务器端的时候，系统根据输入的消息中包含的端点信息将该消息发送给相应的服务器。运行时库将消息传递给服务器过程，由服务器过程将参数反列集，然后调用服务器。应答消息的返回过程正好相反。

DCE 提供了一些语义选项。默认的选择是**至多一次操作**(at-most-once operation)，这种情况下同一个调用的执行不能超过一次，即使系统崩溃时也是如此。在实践中，这意味着如果服务器在 RPC 过程中崩溃，随后又迅速恢复，客户不再重复刚才的操作，因为担心该操作可能已经执行过一次了。

作为另一种解决方法，也可以把某个远程过程标记为**幂等的**(idempotent)(在 IDL 文件中声明)，这意味着可以重复多次执行该过程，而不会造成任何不良后果。例如，从文件中读取指定的某个块的操作就可以反复尝试多次，直到成功为止。如果由于服务器崩溃而导致一个幂等的 RPC 失败，那么客户可以等待服务器重启完毕，然后再次进行尝试。还有其他

一些语义(但是极少使用),包括对本地网络上的所有机器进行 RPC 广播等。在第 8 章中讨论 RPC 的故障问题时会继续探讨 RPC 语义。

4.3　面向消息的通信

远程过程调用和远程对象调用都有助于隐藏分布式系统中的通信,也就是说增强了访问透明性。不幸的是,这两种机制并不总是适用的。特别是当无法保证发出请求时接收端一定正在执行的情况下,就必须有其他的通信服务。同时,RPC 和 RMI 的同步特性也会造成客户在发出的请求得到处理之前被阻塞,因而有时也需要采取其他办法。

这里所说的"其他方法"就是消息传递机制。在本节中,我们重点讨论分布式系统中面向消息的通信。首先详细讨论同步方式的本质及其含义;随后讨论在通信过程中参与通信的各方都处于执行状态的消息传递系统;最后分析消息队列系统,即使通信的另一方在通信开始时并未执行,该系统也能允许进程互相交换信息。

4.3.1　面向消息的瞬时通信

很多分布式系统和应用程序直接构建在由传输层提供的简单的面向消息模型之上。面向消息的系统是中间件解决方案的一部分,为了更深入地理解和评价它,我们首先讨论通过传输层套接字进行的消息传递。

4.3.1.1　Berkeley 套接字

人们已经对传输层接口的标准化给予了充分的重视,以便使程序员通过一个简单的原语集就可以使用传输层提供的全部(消息传递)协议。同时,标准化的接口还使得不同机器之间的应用程序移植变得容易。

作为一个例子,我们将对 Berkeley UNIX 中引入的**套接字接口**(socket interface)进行简要讨论。另一种重要接口是 **XTI**,它是 **X/开放传输接口**(X/open transport interface)的缩写。它的正式名称是**传输层接口**(transport layer interface,TLI),是由 AT&T 公司开发的。套接字与 XTI 采用类似的网络编程模型,但是它们的原语集有些差异。

从概念上说,**套接字**(socket)是一种通信端点。如果应用程序要通过底层网络发送某些数据,可以把这些数据写入套接字,然后再从套接字读出数据。对应于每一种特定的传输协议,本地操作系统都要使用一个实际的通信端点,而套接字形成了位于实际通信端点之上的一个抽象层。在下文中,我们将重点讨论用于 TCP 协议的套接字原语,如图 4.14 所示。

服务器一般执行前 4 个原语,而且一般按照图中给出的顺序执行。调用套接字原语的时候,调用者创建一个新的通信端点,该端点是用于某种特定的传输协议的。从内部来说,创建一个通信端点意味着本地操作系统储备一定的资源,以供特定协议在发送及接收消息时存放消息使用。

bind 原语将本地地址与新创建的套接字相关联。例如,服务器应该将其所在机器的 IP 地址以及一个(可能是公开的)端口号绑定到套接字上。所执行的绑定操作将告知操作系

原语	含义
socket	创建新的通信端点
bind	将本地地址附加到套接字上
listen	宣布已准备好接受连接
accept	在收到连接请求之前阻塞调用方
connect	主动尝试确立连接
send	通过连接发送数据
receive	通过连接接收数据
close	释放连接

图 4.14　TCP/IP 套接字原语

统，服务器希望只在指定的地址和端口上接收消息。

listen 原语只能在面向连接的通信中调用。这是一种非阻塞的调用，它允许本地的操作系统保留足够大小的缓冲区来存放用户希望接受的最大数目的连接。

对 accept 的调用将阻塞调用者，直到有连接请求到达为止。当请求到达的时候，本地操作系统将创建一个与原先的套接字属性相同的套接字，随后将新套接字返回给调用者。这种方法可以让服务器派生出一个新进程，由它来处理新连接的实际通信过程；在此期间服务器则返回到原来的状态，并等待原先套接字上新的连接请求。

我们现在来看一下客户端的情况。同样，首先必须使用 socket 原语创建一个套接字，但是不必将套接字显式绑定到本地地址上，这是因为操作系统可以在连接建立的时候为其动态分配端口。connect 原语要求其调用者指定一个传输层地址，以便将连接请求发送到该地址。调用 connect 后客户被阻塞，直到连接成功建立为止。随后双方使用 write 和 read 原语开始进行信息交换，其中 write 原语用来发送数据，而 read 原语则用来接收数据。最后要关闭连接。使用套接字来关闭连接是一个对称的过程，只有客户和服务器都调用了 close 原语，连接才真正关闭。客户和服务器通过套接字进行面向连接通信的一般模式如图 4.15 所示。在 UNIX 环境下使用套接字以及其他接口进行网络编程的详细资料可以在文献(Stevens 1998)中找到。

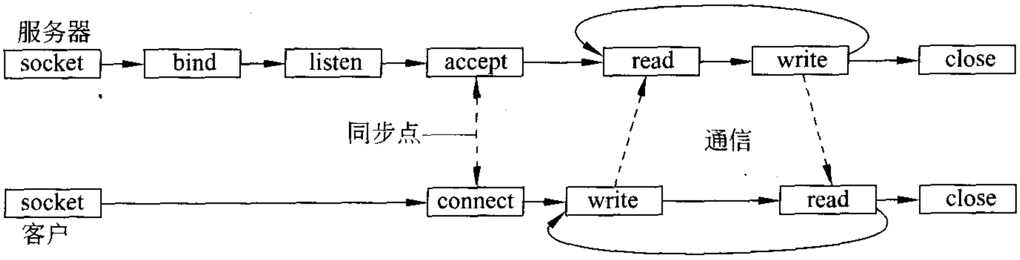


图 4.15　使用套接字的面向连接通信模式

4.3.1.2　消息传递接口(MPI)

随着高性能多计算机系统的出现，开发人员开始寻求有助于更加方便地编写高效率应用程序的面向消息原语。这就要求原语应该提供一个位于适当层次上的抽象（以便简化应用程序开发），而且实现这些原语所花费的开销应该尽量小。由于两方面的原因，套接字被

认为不能满足要求。首先，套接字所在的抽象层不对，它只支持简单的 send 以及 receive 原语。其次，套接字被设计为利用通用的协议栈（例如 TCP/IP）进行网络通信，而并不适用于为高速互联网络开发的专用协议。这些协议要求接口能够提供一些更为高级的特性，比如不同的缓冲和同步方式。

因此，多数互联网络和高性能多计算机系统都附带有专用通信库。这些库提供了丰富的高层通信原语，而且这些原语通常是高效率的。当然，这些库互不兼容，导致应用程序开发人员需要面对程序的可移植性问题。

程序的硬件独立性的需要最终导致了消息传递方面标准的出台。该标准称为**消息传递接口**（message-passing interface，MPI）。MPI 是为并行应用程序设计的，因而是为瞬时通信量身定做的。它直接使用的是底层网络。同时，它还假定诸如进程崩溃或者网络分区之类的严重故障都是致命的，因此不要求这些故障能够自动恢复。

MPI 假定通信在一个已知进程组内发生。每个组都分配有一个标识符（ID），而组内的每个进程也分配有一个局部标识符。因此，一个（组 ID，进程 ID）对就可以唯一地确定消息的来源或者目的地，可以用它来代替传输层地址。在一次计算过程中也许会涉及多个可能相互重叠的进程组，它们在相同时间执行。

MPI 的核心是一些消息传递原语，最直观的一部分原语列在图 4.16 中。

原语	含义
MPI_bsend	将要送出的消息追加到本地发送缓冲区中
MPI_send	发送消息，并等待直到消息复制到本地或远程缓冲区中为止
MPI_ssend	发送消息，并等待直到对方开始接收为止
MPI_sendrecv	发送消息，并等待直到收到应答消息为止
MPI_isend	传递要送出的消息的引用，随后继续执行
MPI_issend	传递要送出的消息的引用，并等待直到对方开始接收为止
MPI_recv	接收消息，如果不存在等待接收的消息则阻塞
MPI_irecv	检查是否有输入的消息，但是无论有没有消息都不会阻塞

图 4.16 MPI 中一些最直观的消息传递原语

瞬时异步通信受 MPI_bsend 原语支持。发送方提交要传输的消息，该消息首先复制到 MPI 运行时系统的本地缓冲区中，消息复制完成后发送方继续执行。本地的 MPI 运行时系统将会把消息从本地缓冲区中移出，并在接收方调用了 receive 原语之后进行消息传输。

也有一个会导致阻塞的发送操作，称为 MPI_send，其语义是依赖于实现的。MPI_send 原语对调用方的阻塞有可能直到指定的消息被复制到发送方一端的 MPI 运行时系统为止，或者直到接收方启动接收操作为止。通过 MPI_ssend 原语，同步通信将发送方阻塞，直到其请求被接收方接受并准备进行进一步处理为止。最后，还支持同步通信的最强形式：当发送方调用 MPI_sendrecv 时，它在向接收方发出请求之后就被阻塞，直到接收方返回应答为止。该原语基本上相当于标准的 RPC。

MPI_send 和 MPI_ssend 各自拥有一种变体形式，使用这些变体时可以避免将消息从用户缓冲区复制到本地 MPI 运行时系统的内部缓冲区中。这两种变体各自对应于一种异步通信形式。如果使用的是 MPI_isend，发送方会传递指向消息的指针，随后由 MPI 运行时系统来管理通信，而发送方则立即继续运行其他程序。为了避免在通信完成之前覆盖消

息，MPI提供了用于检查通信是否完毕的原语，甚至在必要时阻塞调用者。如果使用的是MPI_send，那么不论消息是真的传送给了接收方还是仅仅由本地MPI运行时系统复制到了内部缓冲区中，它都不做说明。

同样，使用MPI_issend的发送方只向MPI运行时系统传递指向消息的指针。在运行时系统告知发送方它已经处理完该消息之后，发送方就能够确认接收方已经接受该消息并正在对它进行处理。

可以调用MPI_recv操作来接收消息，这时，调用者将被阻塞，直到有消息送达为止。它也有一种采用异步方式的变体，称为MPI_irecv，接收方调用MPI_irecv就说明它已经准备好接受消息。接收方可以检查是否确实有消息送达，或者一直保持阻塞状态直到有消息送达为止。

MPI通信原语的语义并不总是简明易懂的。不同的原语有时可以相互替换使用，而不会影响程序的正确性。对于MPI为何支持如此之多的通信方式的官方解释是，这可以赋予MPI系统的开发实现者充分的机会来进行性能优化。批评者认为是委员会无法统一大家的意见，于是把MPI变成了包罗万象的大杂烩。MPI是为高性能并行应用程序设计的，了解了这一点就会明白MPI中的通信原语为何如此多样。

关于MPI的更多资料可以参见（Gropp 等 1998b）。而在（Snir 等 1998）和（Gropp 等 1998a）中有关于MPI的详细解释，其中有上百个MPI函数。

4.3.2 面向消息的持久通信

现在要讨论一类重要的面向消息的中间件服务，一般称为**消息队列系统**（message-queuing system），或者**面向消息的中间件**（message-oriented middleware，MOM）。消息队列系统为持久异步通信提供多种支持。这类系统的本质是，提供消息的中介存储能力，这样就不需要消息发送方和接收方在消息传输过程中都保持激活状态。与Berkeley套接字及MPI的重要区别之一在于，消息队列系统的设计目标一般是支持那些时间要求较为宽松的消息传输，比如那些要求几分钟完成的传输，而不适用于那些必须在几秒内甚至几微秒内完成的传输。我们首先解释消息队列系统的一般实现方法，然后通过把它们与较为传统的系统（如因特网的电子邮件系统）相比较来结束本节的内容。

4.3.2.1 消息队列模型

消息队列系统背后的基本思想是，应用程序可以通过在特定队列中插入消息来进行通信。消息由一系列通信服务器依次进行转发，最终送达目的地。即使在消息发送过程中接收方的机器未处于运行状态，消息也能送到。在实践中，多数通信服务器彼此直接相连，也就是说消息一般直接传递到目的服务器。原则上，每一个应用程序都拥有归其私有的消息队列，其他应用程序可以发送消息到该队列中。队列只能由相应的（也就是拥有该队列的应用程序）读取，但是也可能有多个应用程序共享单个队列。

消息队列系统的重要特征之一是，通常只能确保发送方发出的消息最终能够插入到接收方的队列中，并不保证消息到达的时间，甚至不保证消息一定会得到读取，这完全由接收方来决定。

这些语义允许使用松散耦合的通信方式。因此在消息进入接收方的消息队列时，接收方不必处于运行状态。同样，当发送方所发送的消息被接收方从消息队列中取出的时候，发送方也不必处于运行状态。发送方和接收方可以彼此完全独立地运行。事实上，某个消息一旦进入到队列中，就将一直保留到从队列中移出为止，而与其发送方或者接收方是否正在运行无关。根据发送方和接收方运行状态的不同，一共有 4 种组合，如图 4.17 所示。

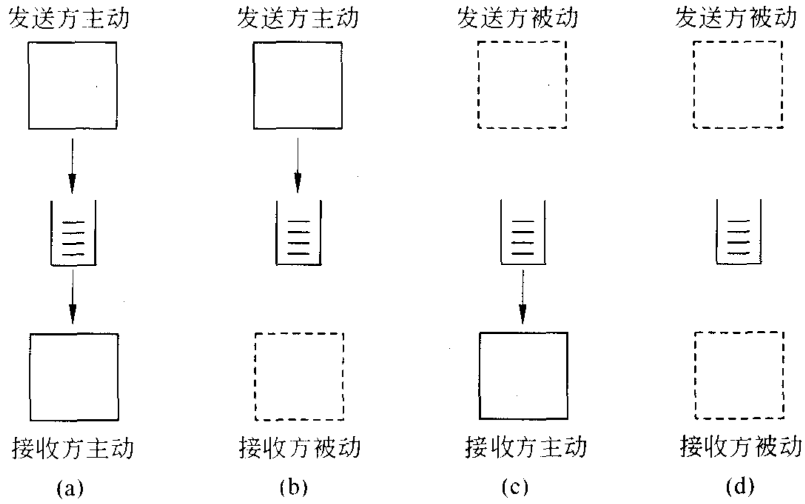


图 4.17 使用队列的松散耦合通信的 4 种组合方式

在图 4.17(a)中，发送方和接收方在消息的整个传输过程中都在运行。在图 4.17(b)中，只有发送方在运行，而接收方并不在运行，也就是说，是处于一种无法进行消息交付的状态，但是发送方仍然可以发送消息。图 4.17(c)中显示的是一种发送方没有运行而接收方在运行的状态。在这种情况下，接收方可以读出发送给它的消息。而在图 4.17(d)中的情况是，虽然发送方和接收方都不在运行，但是消息可以由系统存储(还可能进行传输)。

原则上，消息中可以包含任何数据。唯一重要的是，应该对消息进行正确的编址。在实践中，寻址是通过给出目的消息队列的名字来实现的，该名字在系统范围内是唯一的。在某些情况下，可能对消息的大小做出限制，不过底层系统也可能会以一种对应用程序完全透明的方式来对较大消息的分段及重组进行管理。采用这种方法将使得向应用程序提供的基本接口变得特别简单，如图 4.18 所示。

原语	含义
put	将消息追加进指定队列
get	调用进程阻塞，直到指定队列变为非空为止，然后取出队列中第一个消息
poll	查看指定队列中的消息，并且取出队列中的第一个消息。不阻塞调用进程
notify	注册一个处理程序，在有消息进入指定队列时调用该处理程序

图 4.18 消息队列系统中队列的基本接口

发送方调用 put 原语把要添加到某个指定队列中的消息传递给底层系统。我们曾经说过，这个调用是非阻塞性的。而 get 原语是一个阻塞性调用，调用 get 原语，获得授权的进程将取出指定队列中等待最久的消息。只有在队列为空的时候，进程才被阻塞。这个调用还有几种变体形式，使用这些变体形式可以在队列中搜索给定的消息，比如可以根据优先级

或者某种匹配模式来搜索。poll 原语是 get 原语的一种非阻塞性的变体形式。调用 poll 时，如果队列为空，或者找不到指定的消息，调用进程将简单地直接继续运行。

最后，多数消息队列系统也允许进程注册一个处理程序，称为**回调函数**（callback function），在有消息进入队列时系统将自动调用该处理程序。回调还可以用于在没有进程运行的时候，自动启动一个进程以从队列中获取消息。这常常是通过在接收方一端驻留一个**守护程序**（daemon）来实现的，该程序对队列进行持续的监视，如果有消息进入就进行相应的处理。

4.3.2.2 消息队列系统的一般体系结构

我们现在来详细讨论一般消息队列系统的外部特征。我们所做的第一项限制是，消息只能够放入发送方的本地队列，也就是与发送方位于同一台机器上的队列，至少也应该是附近的机器（比如同一个局域网中的机器）上的队列。这种队列称为**源队列**（source queue）。同样，只能从本地队列中读出消息。然而，放入队列的消息中将会包含对其将要传输到的**目标队列**（destination queue）的说明。由消息队列系统负责向发送方和接收方提供队列，并对消息由源队列向目的队列传输过程进行管理。

意识到下面这点很重要：全部队列的集合是分布在多台机器上的。因此，对于要传输消息的消息队列系统来说，它应该维护一个从队列到其所在网络位置之间的映射关系，这在实践中也就意味着它应该维护一个（也许是分布式的）数据库，其中存储到网络位置所对应的**队列名**（queue name），如图 4.19 所示。应该注意到，这种映射关系与在因特网上用来传输电子邮件的**域名解析系统**（domain name system，DNS）是完全类似的。比如说，如果向逻辑邮件地址 steen@cs.vu.nl 发送邮件，邮件系统将会请求 DNS 找到接收方的邮件服务器所在的网络地址（也就是 IP 地址），服务器用该地址来进行实际的消息传输。

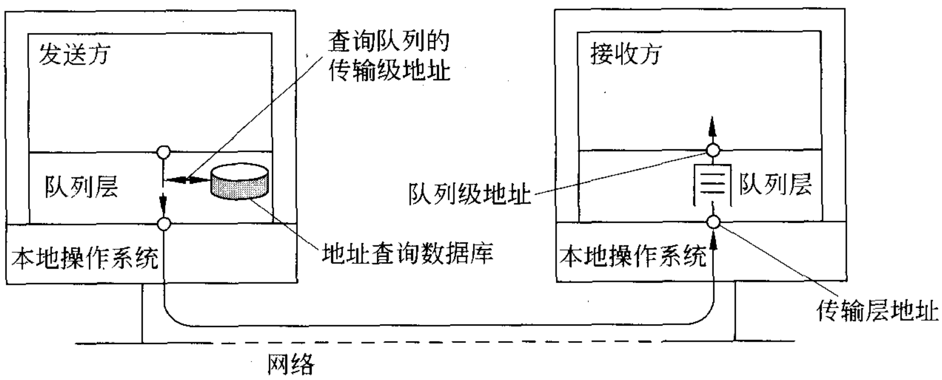


图 4.19 队列级编址与网络级编址之间的关系

队列由**队列管理器**（queue managers）来管理。一般说来，队列管理器与发送或者接收消息的应用程序直接交互。然而，也有一些特殊的队列管理器发挥了路由器或者**中继器**（relay）的作用：它们将输入的消息转发给其他的队列管理器。通过这种方式，消息队列系统可以逐渐演变成为完整的应用级**覆盖网络**（overlay network），该覆盖网络位于原来的计算机网络之上。这种方案与因特网早期的 MBone 架构方案相似，在 MBone 中把普通用户进程都设定为多播路由器。如今，有许多路由器都已经可以支持多播，这就使得覆盖多播变得不那么重要了。

由于多种原因，使用中继器可以带来方便性。比如说，在很多消息队列系统中，没有能对队列-位置映射进行动态维护的通用命名服务。而如果队列网络的拓扑是静态的，每个队列管理器就都需要一份队列-位置映射的副本。不用说，在大规模队列系统中，这种方案很容易导致网络管理方面的问题。

一种解决方案是，使用若干了解网络拓扑的路由器。如果发送方A将目的地为B的消息放入其本地队列中，该消息将会首先传输到最邻近的路由器，在图4.20中是R1。在这里，路由器R1知道如何将该消息往B的方向转发。比如说，R1可以从B的名字推断出应该将消息转发给路由器R2。采用这种方式，只有在路由器需要添加队列或者删除队列时进行更新操作，而其他队列管理器都只需要知道最邻近的路由器所在的位置即可。

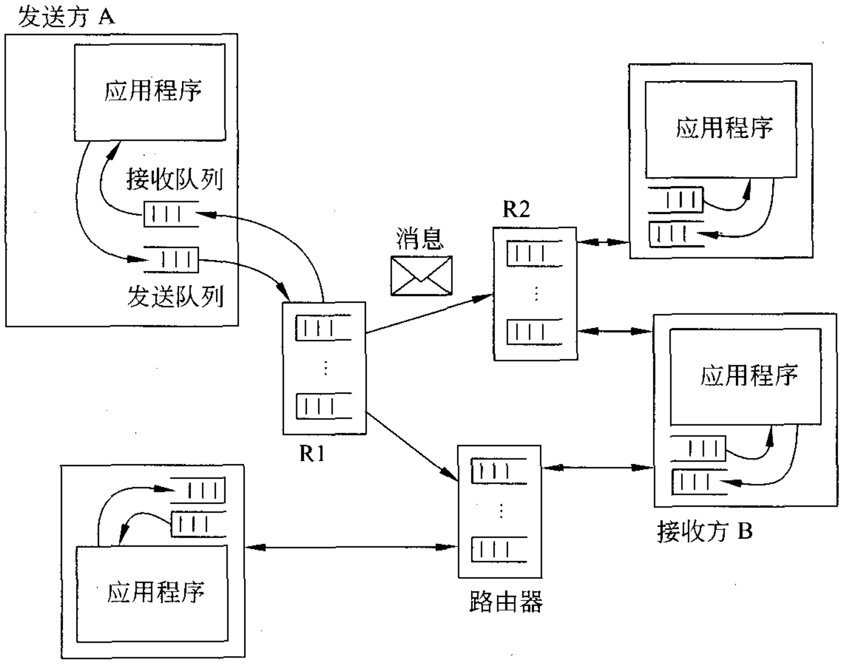


图4.20 含有路由器的消息队列系统的一般组织结构

中继器一般有助于构建可扩展的消息队列系统。然而，显而易见的是，随着队列网络规模的迅速增长，如果还用人工来配置网络，网络很快就会变得难以管理。唯一的解决方法是采用动态路由，这种方法在计算机网络中早已采用了。令人惊讶的是，某些流行的消息队列系统并没有采用这种解决方法。

使用中继器的另一个原因是，中继器允许对消息进行二次处理。比如说，基于安全方面或者容错的考虑，可能要在日志中记录消息。我们将在下一节中讨论中继器的一种特殊形式，它发挥了网关的作用，将消息转换成接收方能够理解的格式。

最后，中继器还可以用于多播的目的。在这种情况下，输入的消息被简单地放入每一个发送队列。

4.3.2.3 消息转换器

消息队列系统的重要应用领域之一就是将现有的应用程序与新应用程序一同集成进一个单一的、结构紧密的分布式信息系统中去。要进行集成，就要求应用程序必须能够理解所接收到的消息，这实际上也就是要求发送方送出的消息所用的格式必须与接收方的消息格

式相同。

这种方法存在的问题是，每一次在系统中添加需要不同消息格式的应用程序时，每个可能的接收方都必须进行改动，以便能够识别新的格式。

另一种方法是，各方达成一致，都采用一种公用的消息格式，就像传统网络协议那样。不幸的是，这种方法在消息队列系统中一般并不适用。问题在系统运行所在的抽象层次上。公用消息格式只在使用该格式的所有进程确实有足够的共同之处的情况下才有意义。如果组成分布式信息系统的应用程序间的差异非常大（这是常有的情况），那么即使最好的公用格式也只不过意味着一串字节序列。

虽然规定了少数供特定的应用程序域使用的公用消息格式，但是一般的方法还是倾向于提供几种共存的不同格式，并提供在各种格式之间尽可能简单地转换手段。在消息队列系统中，转换是由队列网络中特定结点完成的，这些结点称为**消息转换器**（message broker）。消息转换器在消息队列系统中扮演了应用层网关的角色，其主要目的在于将输入消息的格式转换为目的应用程序能够理解的格式。要注意，对于消息队列系统来说，消息转换器也只不过是一个应用程序而已，如图 4.21 所示。换句话说，一般不认为消息转换器是队列系统的一个主要部分。

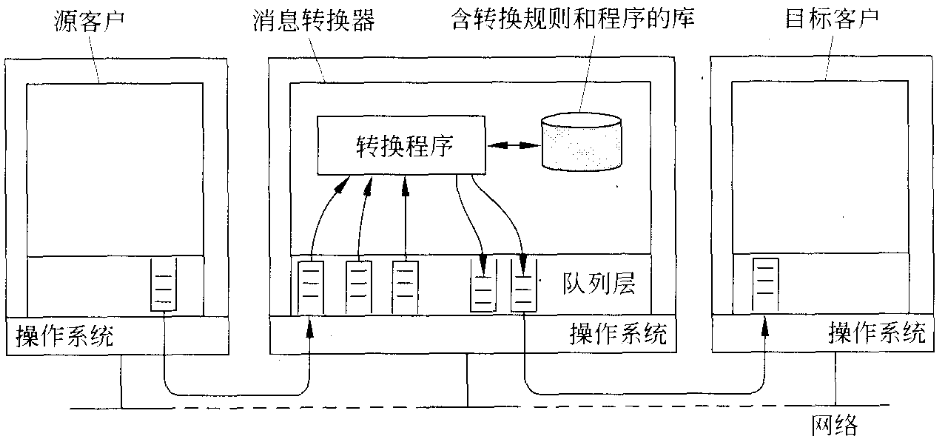


图 4.21　消息队列系统中消息转换器的一般结构

消息转换器可以像一个消息格式重新编排工具一样简单。比如说，假定输入的消息包含有数据库中的一张表，其中，记录之间通过特殊的代表记录结束的分隔符来分隔，而记录中的字段长度已知且固定。如果目的应用程序希望使用另外的分隔符来分隔记录，并且希望字段是可变长度的，就需要使用消息转换器来将消息的格式转换为目的应用程序所能够接受的格式。

拥有更高级设置的消息转换器可以扮演应用层网关的角色，比如处理 X.400 消息格式与因特网电子邮件格式之间的转换。在这种情况下，常常不能够保证输入消息中包含的所有信息都能够实际转换为输出消息中相对应的内容。

但更为常见的是为高级的**企业应用集成**（enterprise application integration，EAI）使用消息转换器，正如我们在第 1 章中所介绍的那样。在这种情况下，除了进行消息转换之外，转换器还要根据正在交互的消息匹配应用程序。这种模型称为**发布/订阅**（publish/subscribe），应用程序以发布的形式来发送消息。它们可能发布一个关于主题 X 的消息，然

后发送给转换器。那些已经说明了它们对主题 X 的消息感兴趣的应用程序(也就是说,它们订购了这些消息)将接收来自转换器的这些消息。可以有更高级的形式,我们将在第 13 章中介绍。

消息转换器的核心是一个数据库,数据库中的规则规定了如何将格式为 T1 的消息转换为格式为 T2 的消息。问题在于如何定义这些规则。大多数消息转换器的产品附带有复杂的规则开发工具,但是仍然要求专家将规则输入数据库。我们看到过很多的示例,那些商业产品往往认为能提供"智能",但实际上智能只能在这些专家的脑海中找到。

4.3.2.4　消息队列系统说明

请回顾一下我们谈论过的关于消息队列系统的内容,看起来好像这种系统早就以电子邮件服务的方式实现了。电子邮件系统一般是通过一组邮件服务器实现的,这些服务器代表与其直接相连的主机上的用户来存储并转发消息。电子邮件可以直接使用底层传送服务,而并不涉及路由问题。例如,在因特网的主要协议 SMTP(Postel 1982)中,可以直接与目的邮件服务器之间建立 TCP 连接来传输消息。

电子邮件系统与消息队列系统相比较,特别之处在于前者的目的主要在于为最终用户提供直接支持。这就说明了为什么有多种群件应用程序是直接基于电子邮件系统的(Khoshafian 和 Buckiewicz 1995)。另外,电子邮件系统可能有一些特殊的需求,例如,自动消息过滤、高级消息数据库支持(比如,为了更方便地取得先前存储的数据的需要)等。

普通消息队列系统的目标并不仅仅在于为最终用户提供支持。建立这样的系统要解决的重要问题是:要为进程间的持久通信提供支持,无论该进程是在运行用户程序,在处理对数据库的访问,还是在进行计算等,都必须能够支持其通信。这就导致消息队列系统的需求与纯粹的电子邮件系统不同。比如说,电子邮件系统一般不需要提供可靠的(有保证的)消息传递、消息的优先级、日志功能、高效的多播、负载平衡、容错性等。

因此,通用的消息队列系统的应用范围很广,包括电子邮件、工作流、组件以及批处理。然而,它最重要的应用领域是将一组数据库(也许分散在广阔区域)或者数据库应用程序集成进多个数据库(信息)系统中去(Hohpe 和 Woolf 2004)。比如说,一个涉及若干个数据库的查询可能需要切分成为几个子查询,然后将这些子查询转发给相应的数据库。在这个过程中可以利用消息队列系统所提供的基本方法来将每个子查询打包进消息中去,然后将消息发送给相应的数据库。在本章中讨论过的其他通信工具在这方面都远没有这么适用。

4.3.3　示例:IBM WebSphere 消息队列系统

为了帮助理解消息队列系统的实际工作方式,我们来研究一个特定的系统,即消息队列系统,它是 IBM WebSphere 产品的一部分。以前称为 MQSeries,现在称为 **WebSphere MQ**。关于 WebSphere MQ 的文档很多,下面我们只是探讨基本原理。关于消息队列网络的很多体系结构细节内容可以参考(IBM 2005b 2005d)。消息队列网络的编程不是用某个星期天下午就能学会的,MQ 编程指南(IBM 2005a)是一个很好读物,它从原理到实践进行讲解,但可能需要花些时间去学习。

4.3.3.1 概述

MQ 队列网络的基础体系结构是非常简单易懂的，如图 4.22 所示。所有队列由**队列管理器**(queue managers)进行管理。队列管理器负责从其发送队列中取出消息，然后将消息转发给其他队列管理器。同样，队列管理器还负责从底层网络获取输入的消息，然后将每个消息存储在恰当的输入队列中。默认情况下，最大消息为 4 MB，但可以增加到 100 MB。队列通常限制在 2 GB 的数据，但根据操作系统的不同，这个最大值可以很容易设得更高些。

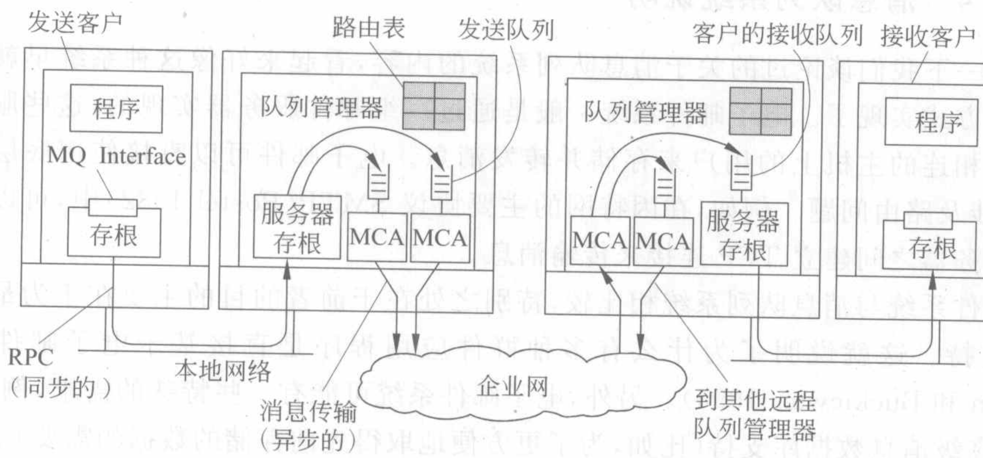


图 4.22 IBM 消息队列系统的一般组织结构

一对队列管理器是通过**消息通道**(message channel)相连的。消息通道是传输层连接的一个抽象。消息通道是发送方与接收方队列管理器之间的一种单向的可靠连接。队列中的消息通过该通道进行传输。例如，基于因特网的消息通道是作为 TCP 连接实现的。消息通道的两端各自由一个**消息通道代理**(message channel agent，MCA)来管理。发送端 MCA 所做的基本上只是：检查发送队列中是否有消息，如果有的话将消息包装成传输层的包，然后将包通过连接发送给对应的接收端 MCA。同样，接收端 MCA 的基本任务只是监听是否有包送达，如果有的话将包解开，随后将由包中得到的消息存储到恰当的队列中。

队列管理器可以连接到其管理的队列所属的应用程序所在的进程中去。在这种情况下，队列通过标准接口对应用程序隐藏，但是应用程序可以有效地对队列进行操纵。另一种组织结构是，队列管理器与应用程序运行在不同的机器上。在这种情况下，向应用程序提供的接口与第一种情况——也就是应用程序与队列管理器位于同一台机器上时相同。然而，不同的是这时接口是作为代理实现的，它使用基于 RPC 的传统同步通信来与队列管理器通信。通过这种方式，MQ 基本保留了这种模型，即只能够访问应用程序的本地队列。

4.3.3.2 通道

消息通道构成了 MQ 的一个重要部分。每个消息通道都拥有且只拥有一个与其相对应的发送队列，它从中取出消息，传输到另一端。只有在通道两端的 MCA——发送端 MCA 和接收端 MCA 都已启动运行的情况下，消息才能沿通道进行传输。除了可以手工启动两端的 MCA 之外，还可以采取某些别的办法来激活一个通道，下面我们将讨论其中的几

种方法。

一种方法是，由应用程序通过激活发送端或者接收端 MCA 来直接启动通道的一端。然而，如果考虑到透明性，这种方法并不十分可取。启动发送端 MCA 的更好的办法是，对通道的发送队列进行配置，使其在第一个消息放入队列时触发一个触发器(trigger)，该触发器随之启动与其相关联的一个处理程序，该处理程序负责启动发送端 MCA，随后就可以由 MCA 从发送队列中取出消息。

还有一种方法是通过网络来启动 MCA。特别是，如果通道一端已经激活，它就可以发送一条控制消息，请求另一端的 MCA 启动。该控制消息是发送给与要启动的另一端 MCA 位于同一台机器上的守护程序的，该守护程序监听某个公开的地址。

如果不再有消息送入发送队列，经过一段时间后通道将会自动关闭。

每一个 MCA 都有一组相关的属性，这些属性决定了通道的全部特性。其中某些属性如图 4.23 所示。发送端 MCA 与接收端 MCA 的属性值应该是一致的，并且应该在建立通道之前就协商好。比如说，两端的 MCA 都应该明确地支持同一传输协议。有的属性值无法进行协商，比如消息是否应该按照放入发送队列的次序来进行传输就是这样的属性。也就是说，如果一端的 MCA 要以 FIFO(先进先出)的次序来传输，另一端就必须遵守这个次序。而如果属性值包含了消息最大长度，则可以采用两端 MCA 对应的该属性值中较小的那一个。

属性	描述
transport type(传输类型)	决定要采用的传输协议
FIFO delivery(先进先出传输)	表明消息将按照发送的次序到达
message length(消息长度)	单个消息的最大长度
setup retry count(建立连接的重试次数)	指定启动远程 MCA 的最大重试次数
delivery retries(消息交付重试次数)	MCA 将收到的消息放入队列的最大重试次数

图 4.23　与消息通道代理相关的一些属性

4.3.3.3　消息传输

为了从一个队列管理器往另一个队列管理器传输消息，每一个消息中都必须包含它的目的地址，也就是说必须使用传输报头。MQ 中的地址由两部分构成：第一个部分包含将要接收消息的队列管理器的名字，而第二部分是目的队列的名字，队列管理器将把消息添加到该队列中去。

除了目的地址之外，还必须指定消息传输的路由。路由的说明是通过给出其本地发送队列的名字来实现的，消息将被添加到该队列中。因此，不必在消息中包含完整的路由。回忆一下，每个消息通道都拥有且只拥有一个发送队列。通过指定消息应添加到哪一个发送队列中，就可以有效地规定应该将消息转发到哪一个邻近的队列管理器。

大多数情况下，路由被显式地存储在队列管理器中的路由表中。路由表中的条目是(destQM，sendQ)对，其中 destQM 是目的队列管理器的名字，而 sendQ 是本地发送队列的名字，队列管理器将把消息放入该队列中去(路由表中的条目在 MQ 中称为别名)。

消息在传输过程中可能需要经过多个队列管理器才能到达目的地。当消息被中途遇到

的队列管理器收到时，该队列管理器简单地从消息报头中提取出目的队列管理器的名字，随后在路由表中进行查询，以决定要将该消息放入哪一个本地发送队列中去。

认识到下面这点是很重要的：队列管理器拥有一个系统范围内唯一的名字，该名字可以有效地用作队列管理器的标识符。但是这样做带来的问题是：替换队列管理器或者更改队列管理器的名字会对向该管理器发送消息的所有应用程序造成影响。这个问题可以通过为队列管理器取一个**本地别名**（local alias）而部分地得到解决。如果由队列管理器 M1 为队列管理器 M2 定义了一个别名，那么只有使用 M1 接口的应用程序可以使用该别名。即使管理某个队列的队列管理器发生了改变，还是可以使用该队列原来的逻辑名来访问该队列。如果队列管理器的名字改变了，就需要改变它在其他所有队列管理器中的名字，但不会对应用程序造成影响。

路由表以及别名的使用规则如图 4.24 所示。例如，连接到队列管理器 QMA 的应用程序可以使用本地别名 LA1 来引用远程队列管理器。QMA 将会首先在别名列表中查询消息实际的目的地，发现该别名代表的是队列管理器 QMC。随后，在路由表中找出到 QMC 的路由，其中指明了要发往 QMC 的消息应该放入队列 SQ1 中，而 SQ1 是用来向队列管理器 QMB 传输消息的。在收到该消息后，QMB 将会利用它自己的路由表将消息转发到 QMC。

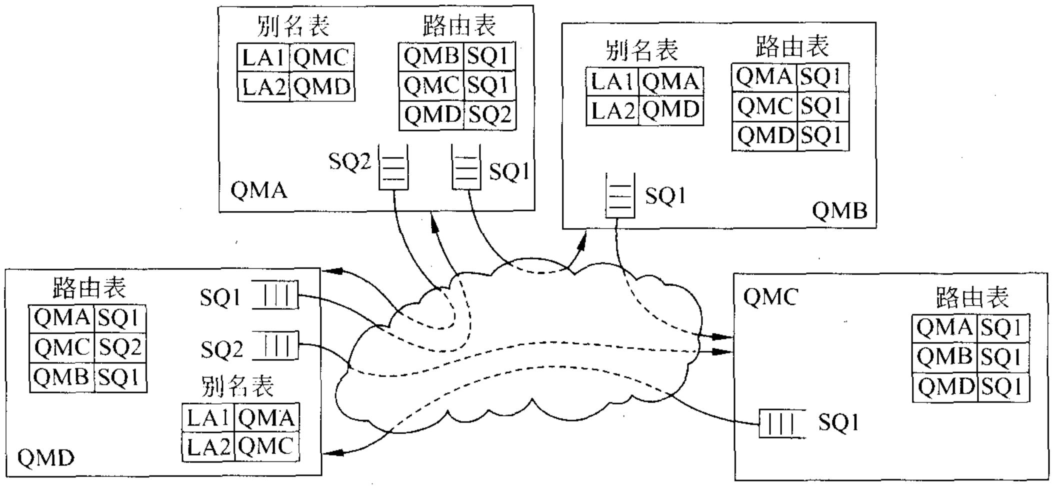


图 4.24 使用了路由表和别名表的 MQ 队列网络的常见组织结构

如果采用这种路由和别名方案，就会使得应用程序编程接口相对简单，这时的接口称为**消息队列接口**（message queue interface，MQI）。MQI 中最重要的原语如图 4.25 所示。

原语	描述
MQopen	开启一个队列（可能位于远程）
MQclose	关闭一个队列
MQput	将消息放入开启的队列中
MQget	从队列（队列可以在本地）中获取消息

图 4.25 消息队列接口中的原语

如果要将消息放入队列中，应用程序就需要调用 MQopen 原语，同时指定特定队列管理器中的某个目的队列。队列管理器可以使用可用的本地别名来命名。目的队列是否位于

远程的问题对应用程序是完全透明的。如果应用程序要从本地队列中取出消息，也需要调用 MQopen。只能够打开本地的队列来读取送达的消息。当应用程序结束对队列的访问之后，需要调用 MQclose 来关闭该队列。

可以调用 MQput 向队列中写入消息，调用 MQget 来从队列中读取消息。理论上说，是根据消息优先级来决定将哪一条消息移出队列的。如果消息优先级相同，就按照先入先出的原则来决定移出哪一条消息，等待时间最长的消息将首先被移出。也可以要求取出某一条特定消息。最后，MQ 提供了在特定消息送达时通知应用程序的功能，这就避免了应用程序为了等待接收某个消息而不断对消息队列进行查询。

4.3.3.4 管理覆盖网络

从以上介绍可知，显然，管理 MQ 系统的一个重要部分是把各种队列管理器连接到一个一致的覆盖网络中。而且，这个网络需要不停地维护。对小型网络来说，这种维护不会超过一般的网络管理工作，但当消息队列用于集成或分解大型的已有系统时，情况就变得复杂了。

MQ 的一个主要问题是，覆盖网络需要手工管理。这种管理不仅包括在队列管理器之间创建通道，还要填写路由表。显然，这将会是一个恶梦。对 MQ 系统的管理支持的进展不大，管理员可以虚拟地设置每个可能的属性，调整可以想到的配置。然而，通道和路由表最终还是得手工维护。

覆盖网络的核心是**通道控制功能**（channel control function）组件，在逻辑上，它位于消息通道代理之间。该组件允许操作人员准确地监视在通道的两端都在进行些什么。而且，可以用它来创建通道和路由表，还可以管理驻留有消息通道代理的队列管理器。在一定程度上，覆盖网络的这种方法很像服务器集群的管理，其中只使用一个管理服务器。在服务器集群中，服务器只是给集群中的每个机器提供一个远程外壳以及一些处理机器组的操作。关于分布式系统管理的一个好消息是，如果你希望探究难题的新的解决办法，它提供了很多机会。

4.4 面向流的通信

到目前为止，我们讨论的通信都是对某种程度上独立且完整的信息单元进行交换。例如，调用某个过程或者方法的请求、对这种请求的响应，以及消息队列系统中应用程序间的消息交换都属于这种类型的通信。这种类型通信的典型特征是，它并不在乎通信究竟在哪个确切时间发生。虽然系统可能会运行得过快或者过慢，但是同步对通信的正确性没有影响。

但是，在某些形式的通信中，同步扮演了关键的角色。考虑一下这种情况：比如一个采用 16 位样本序列表示的音频流，每一个样本通过脉冲码调制（pulse code modulation，PCM）来表示声波的振幅。假设该音频流需要达到 CD 音质，也就是说应该对原始声波进行频率为 44 100Hz 的采样。为了复现原声，关键在于音频流中的样本必须按照次序来播放，而且播放的每两个样本之间的时间间隔必须为严格的 1/44 100s。如果以别的速率来播放，

就会造成播放的声音与原声不同。

我们在本节中要讨论的问题是，分布式系统应该提供怎样的功能，来为时间敏感的信息交换（比如音频流和视频流）提供支持。（Halsall 2001）中讨论了用来处理面向流通信的协议。而（Steinmetz 和 Nahrstedt 2004）对多媒体方面内容进行了综合介绍，面向流的通信是其中的一部分。有关对数据流的查询处理参见（Babcock 等 2002）。

4.4.1 为连续媒体提供支持

为时间敏感的信息交换提供支持一般表示为连续媒体提供支持。在这里，媒体是指传送信息的手段，其中包括存储以及传输介质、如监视器之类的显示媒介等。媒体的重要类型是信息的表示方式，也就是信息在计算机系统中的编码方式。对于不同类型的信息要使用不同的表示形式。比如说，文本一般用 ASCII 或者 Unicode 来进行编码。图像可以采用 GIF 或者 JPEG 等格式来表示。而计算机系统中的音频流可以采用诸如 PCM 的 16 位样本来编码。

在**连续（表示）媒体**（continuous (representation) media）中，了解不同数据项之间在时间上的联系是正确解释数据含义的基础。我们已经给出过这样的例子：通过播放某个音频流以复现原来的声音。作为另一个例子，我们来考虑一下运动。运动可以通过一系列图像来表现，这些图像必须以相同的间隔时间 T 来连续显示，T 一般为 30～40ms。如果要正确地表现运动过程，不但要以正确的顺序显示图像，还要以恒定的显示速率——每秒 1/T 幅图像来显示。

与连续媒体相反，**离散（表示）媒体**（discrete (representation) media）的特征是：数据项之间的时间联系对于正确解释数据含义并不重要。离散媒体的典型例子包括文本、静态图像的表示，还有对象代码及可执行文件等。

数据流

为了对时间敏感的信息进行交换，分布式系统一般都提供对**数据流**（data stream）的支持。数据流是数据单元的序列，可以应用于离散的媒体，也可以应用于连续媒体。比如说，UNIX 中的管道或者 TCP/IP 连接就是面向字节的离散数据流的例子。而播放音频文件时一般要求在文件与音频设备之间建立连续数据流。

对于连续数据流来说，同步是极为关键的。为了捕捉同步状况，要对不同的传输模式做出区分。在**异步传输模式**（asynchronous transmission mode）下，流中的数据项是逐个传输的，但是对某一项在何时进行传输并没有进一步的限制。这是采用离散数据流时常见的情况。比如说，文件可以作为数据流来传输，但是每一数据项传输完成的确切时间通常是无关紧要的。

在**同步传输模式**（synchronous transmission mode）下，数据流中每一个单元都定义了一个端到端最大延迟时间。数据单元的传输时间是否远远小于最大允许延迟并不重要。比如说，传感器以某个特定的采样率对温度值进行采样，然后将采样结果通过网络传给操作员。在这种情况下，很重要的一点是必须确保端到端的网络传输时间小于采样间隔时间，但是如果采样结果的传输速度大大超过所需的最低值是没有关系的。

最后，还有一种**等时传输模式**(isochronous transmission mode)，在这种模式中数据单元必须按时传输，也就是数据传输的端到端延迟时间必须同时受到上限和下限的约束，端到端延迟时间上限和下限也称为边界延迟抖动。等时传输模式在表现视频和音频方面扮演了关键角色，因此它对分布式多媒体系统极为有用。在本章中，我们只考虑使用等时传输的连续数据流，简称为流。

流可以很简单，也可以很复杂。一个**简单流**(simple stream)只包含有单个数据序列，而一个**复杂流**(complex stream)由若干相关的简单流——**子流**(substream)构成。复杂流中各子流之间的关系常常是时间敏感的。比如说，立体声音频可以使用包含两个子流的复杂流来传输，其中每个子流代表一个声道。然而，这两个子流必须是始终保持同步的，这一点很重要。换句话说，来自两个流的数据单元都必须成对传输，以确保产生立体声的效果。另一个复杂流的例子是关于影片传输的。这种流由单个视频流，以及两个用于以立体声方式传输电影中声音的音频流。第 4 个流用来传输字幕或者翻译成其他语言的文字。这里，各子流间的同步依然是重要的。如果无法做到同步，就无法完整地复现影片。我们将在稍后继续讨论这个问题。

从分布式系统的角度，可以区分支持流所需的各种元素。为简单起见，这里只关注已存储数据与原始数据的流传输。在后一种情况下，数据被实时捕捉并在网络上发送给接收方。这两种情况的主要差别是，原始数据的流传输留给调整流的机会少。按照(Wu 等 2001)，我们可以为连续多媒体数据流描述一个一般的客户-服务器体系结构，如图 4.26 所示。

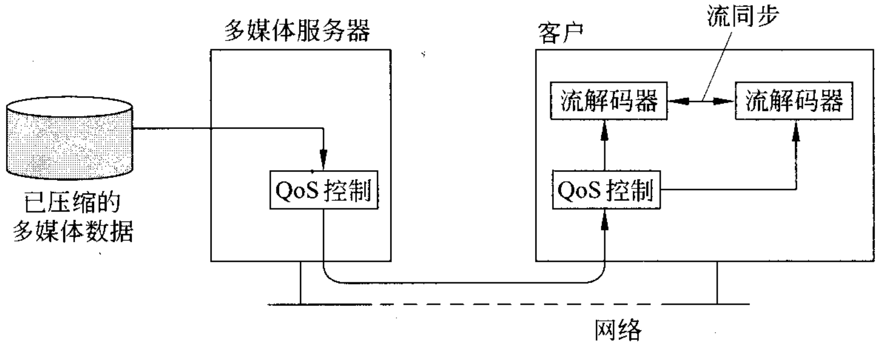


图 4.26　用于已存储的多媒体数据的流传输的客户-服务器体系结构

这个体系结构揭示了很多需要处理的重要问题。首先，多媒体数据，尤其是视频和音频，需要充分压缩以便减少所需的存储空间，特别是网络容量。从通信的角度来看，更为重要的是控制传输的质量和同步化问题。我们将在后面讨论这个问题。

4.4.2　流与服务质量

时间敏感的(以及其他非功能性的)需求一般统称为**服务质量**(quality of service，QoS)需求。这种需求描述了底层分布式系统及网络在确保传输质量方面的需要。连续数据的 QoS 主要关注时间、容量以及可靠性。在本节中我们将详细讨论 QoS 以及它与创建流之间的关系。

很多文章介绍了如何描述 QoS 需求(例如，Jin 和 Nahrstdt 2004)。从应用程序的角度

来看，在很多情况下，可以简化为指定如下一些特性(Halsall 2001)：

(1) 数据传输所要求的比特率。

(2) 创建会话的最大延时(例如，应用程序何时可以开始发送数据)。

(3) 端到端的最大延时(例如，数据单元到达接收端花费多少时间)。

(4) 最大延时抖动。

(5) 最大往返延时。

注意，可以对这些特性进行很多的改进，正如(Steinmetz 和 Nahrstadt 2004)所介绍的那样。但是，当处理基于因特网协议栈的面向流的通信时，我们必须接受这样一个事实：通信的基础是由一种非常简单且卓有成效的数据报服务 IP 形成的。现在，很多(尽管不是全部)支持面对流通信的分布式系统都是构建在因特网协议栈的上面。关于 QoS 说明就这么些(实际上，IP 是提供了一些 QoS 支持，但很少被实现)。

确保 QoS

如果底层系统只提供传送服务，分布式系统就可以尽可能地隐藏掉服务质量的缺乏。这里有多种机制可以部署。

首先，情况并不像到目前为止所描述的那样糟糕。例如，因特网通过它的**区分服务**(differentiated services)，为不同类型的数据提供服务。发送主机可以把输出数据包标记为属于哪一类型，包括**快速转发**(expedited forwarding)类型。快速转发类型指定当前路由器应以绝对优先级来转发数据包(Davie 等 2002)。另外还有一种**确保型服务**(assured forwarding)，其中的数据流量分为四种子类，如果网络发生拥塞，有三种方式来丢弃数据包。因此，确保型服务定义了各种优先级，这些优先级可以赋给数据包，从而允许应用程序把时间敏感的数据包与无关紧要的数据包区分开来。

除了这些网络层面的解决办法，分布式系统还可以帮助接收方获得数据。尽管可用的工具并不多，但有一种特别有用，它使用缓冲区来减少抖动。其原理很简单，如图 4.27 所示。假设当数据包在网络上传输时有不同的延时，接收方先把它们存储在缓冲区中。这样，知道总是有足够的数据包进入缓冲区，接收方就可以以固定的速率把数据包传递给应用程序。

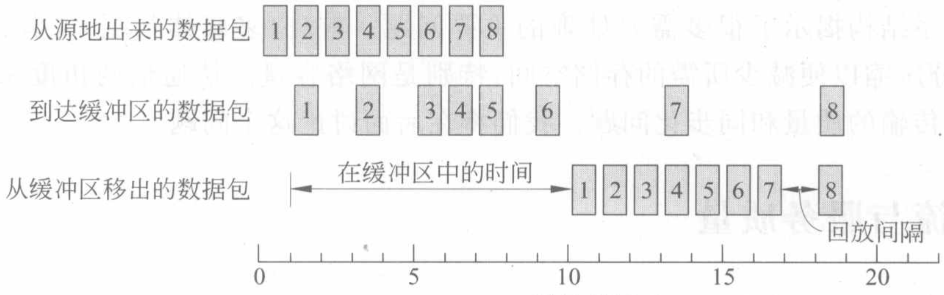


图 4.27 使用缓冲区来减少抖动

当然，事情可能会变得糟糕，如图 4.27 所示的第八个数据包。接收方缓冲区的大小对应于数据包传递给应用程序的 9 秒钟。但是，第八个数据包到达接收方花费了 11 秒钟，此时，缓冲区已完全空了。这就导致出现应用程序的回放间隔。唯一的解决办法是增加缓冲

区的大小。其明显的缺陷是，接收应用程序可以开始回放缓冲区中的数据包的延时同样增加了。

还可以使用其他技术。我们知道，底层的有效服务也可能丢失数据包。为了从服务质量上补偿这种丢失，需要应用纠错技术(Perkins 等 1998；Wah 等 2000)。通常，要求发送方重传丢失的数据包是不可能的，因此需要应用**转发纠错**(forward error correction)技术。一种著名的技术是以这样一种方式给输出数据包编码：n 个中接收了 k 个数据包，就足以重构这 k 个数据包。

这里可能出现的一个问题是，单个的数据包可能含有多个音频和视频帧。因此，当某个数据包丢失，接收方在播放帧时会发现一个较大的间隔。这种情况可以通过交错传输来避免，如图 4.28 所示。以这种方式，当某个数据包丢失时，在连续多个帧中导致的间隔分布较广。但要注意的是，与非交错传输相比，这种方法要求有更大的接收缓冲区，因此使得接收应用程序的开始延时更高。例如，考虑图 4.28(b)，要播放前 4 帧，接收方需要等到 4 个数据包都已传送，而不是像非交错传输那样只需一个数据包。

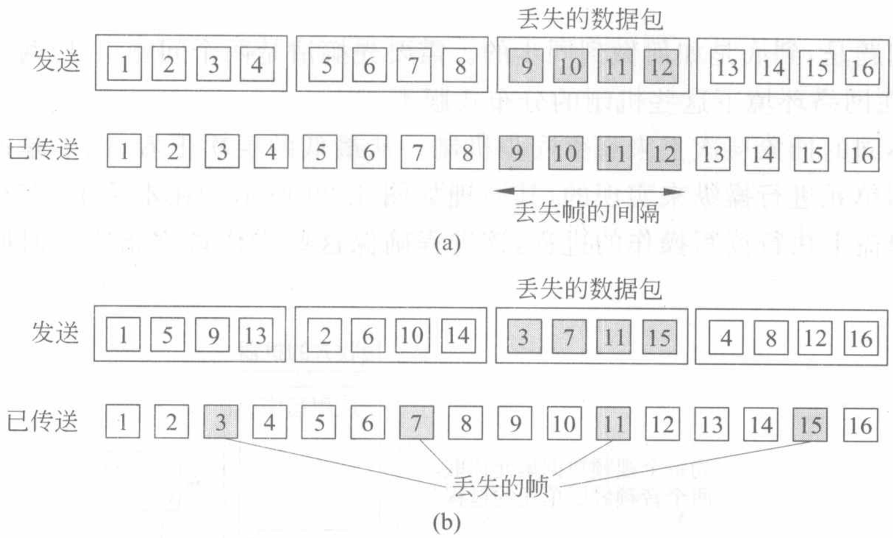


图 4.28　数据包丢失的结果

(a) 非交错传输；(b) 交错传输

4.4.3　流同步

多媒体系统中一个重要的问题是，不同的流(可能都是复杂流)，可能要互相保持同步。流同步涉及的问题是要在流之间保持时间上的关联，流同步有两种类型。

最简单的一种同步是在离散数据流与连续数据流之间保持同步。我们来考虑一下 Web 上的带有音频的幻灯片放映。幻灯片以离散数据流的形式由服务器传输给客户，而同时客户要播放特定的(一部分)音频以配合当前放映的幻灯片，而音频也是从服务器取得的。在这种情况下，音频流必须与幻灯片的演示保持同步。

要求更加苛刻的一类同步是连续数据流之间的同步。日常生活中就有这方面的例子：放映影片时视频流必须与音频流保持同步，这常常称为口型同步。另一个例子是放送立体声。立体声音频流由两个子流组成，每个子流代表一个声道。要正确地表现立体声，就要求

两个子流保持紧密的同步：只要子流间相差 20μs 就足以导致立体声失真。

同步是在建立流的数据单元这个层次上的。也就是说，只要让两个流的数据单元保持同步，就可以让两个流同步。对数据单元的选择高度依赖于看待数据流的抽象层次。让我们来更具体地考虑一下前面提过的 CD 音质的（单声道）音频流。在最适当的粒度上，这种流呈现为 16 位样本序列。如果采样频率为 44 100Hz，在理论上每约 23μs 就必须进行一次同步。实践证明，如果要达到高质量的立体声效果，这种级别的同步确实是必需的。

然而，当我们考虑音频流与视频流之间的口型同步时，就必须采用比较粗糙的粒度。我们曾经说过，视频帧必须以 25 Hz 以上的速率显示。广泛采用的 NTSC 标准规定的是 29.97Hz，如果采用这个标准，我们就必须将音频样本组成逻辑单元，音频样本的长度必须与视频帧的显示时间相同（33ms）。如果音频采样频率为 44100Hz，那么音频数据单元就必须包含 1470 个样本，也就是 11 760 字节（假定样本为 16 位）。在实践中，可以允许更大的长度为 40ms 或 80ms 的单元（Steinmetz 1996）。

同步机制

现在的问题是，到底是如何做到同步的。需要先搞清楚两个问题：（1）两个流同步的基本机制；（2）在网络环境下这些机制的分布式版本。

可以在不同的抽象层次上来观察同步机制。从最低的层次上看，同步是通过显式地对单个流的数据单元进行操纵来实现的，其原理如图 4.29 所示。在本质上，存在一个专门在少数几个简单流上执行读写操作的进程，该进程确保这些操作遵守指定的时间及同步约束条件。

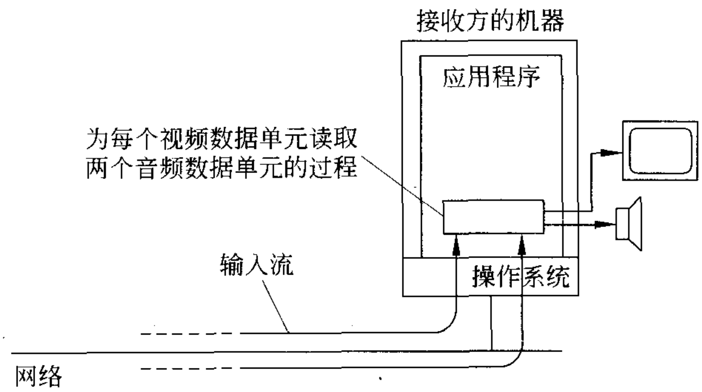


图 4.29 在数据单元层次上进行显式同步的原理

例如，我们来考虑一下一部用两个输入流来表示的影片。视频流中包含有未经压缩的低质量图像，分辨率为 320×240 像素，每个像素由一个字节表示，这样的图像中每个视频数据单元的大小为 76800 字节。假定图像要以 30 Hz 的频率显示，那么每幅图像就要持续 33ms。假定音频流包含的音频样本组成了大小为 11760 字节的单元，那么每个单元就相当于 33ms 的音频，这些我们在前面已经讨论过了。如果输入进程的输入速率为 2.5 MB/s，只要每 33ms 读取一次图像并读取一次音频数据块就可以做到口型同步了。

这种方法的缺陷在于，完全由应用程序来负责实现同步，但它只有低层功能可用。更好的做法是，向应用程序提供接口，允许它更加方便地对流和设备进行控制。回到上一个例子，假定视频显示有一个控制接口，允许指定图像显示的速率。该接口还提供注册用户处理

程序的功能，每当有 k 个新图像到达的时候该用户处理程序就被调用。音频设备也可以提供相似的接口。通过这些控制接口，应用程序开发人员可以编写出简单的监控程序，它由两个处理程序组成，每个处理程序负责处理一个流，并由两个处理程序相配合，共同检查视频流和音频流是否完全同步，并且在必要的时候调整视频或音频单元的显示或播放速率。

最后一个例子如图 4.30 所示，它对于许多多媒体中间件系统来说是很典型的。多媒体中间件提供了一组接口用于控制视频流和音频流，其中包括控制诸如显示器、照相机和麦克风等设备的接口。每个设备和每个流都有自己的高级接口，其中包括在发生某些事件时通知应用程序的接口，后者用来编写用于流同步的处理程序。在(Blair 和 Stefani 1998)中给出了这种接口的例子。

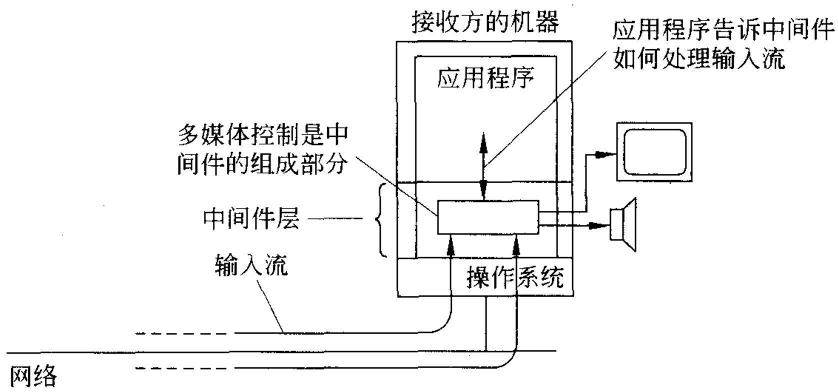


图 4.30 通过高级接口支持同步的原理

需要注意的另一个问题是，同步机制的分布性。首先，由请求同步的子流组成的复杂流的接收方需要了解同步究竟要达到哪些要求。也就是说，它必须使完整的同步说明(synchronization specification)在本地可用。在实践中一般是隐式地将不同流多路复用到包含所有数据单元(其中含用于同步的数据)的单个流中。

MPEG 流采用了另一种同步方案。**活动图像专家组**(motion picture experts group, MPEG)标准定义了一组用于视频及音频压缩的算法。现在已经有了若干种 MPEG 标准。比如 MPEG-2，该标准原来是用来将广播级质量的视频压缩到 4～6Mb/s。在 MPEG-2 中，可以将数目不限的连续流和离散流合并到单个流中。每个输入流先转换成包的流，包中带有由基于 90 kHz 的系统时钟给出的时间戳，这些包的流又被多路复用为一个**节目流**(program stream)，节目流中的包长度是可变的，而这些包的共同点在于其带有相同的时基。在接收端需要对节目流进行多路分离，此时再一次利用包中的时间戳作为流间同步的基本机制。

另一个重要的问题是，应该选择在发送方还是接收方进行同步。如果由发送方来处理同步，就可以将许多流合并成单个流，并且使用另一种类型的数据单元。让我们来再次考虑一下前面例子。例子中，由两个各代表一个声道的子流构成了立体声音频流。一种可能采取的做法是将两个子流彼此独立地传输给接收方，由接收方进行两个流之间的同步。很明显，由于每个子流的延迟可能不同，要做到同步是非常困难的。更好的做法是在发送方将两个子流合并起来，得到的流中的数据单元是由成对的样本构成的，分别来自左右声道的两个样本组成一对，这样接收方只要读取数据单元并将它分离成左声道样本和右声道样本就行了，在这种情况下，两个声道的延迟相同。

4.5 多播通信

在分布式系统通信中的一个重要主题是对多个接收方发送数据的支持，也称为多播通信。在很长一段时间里，这个主题属于网络协议的研究领域，其中已实现和发展了网络层和传输层解决办法的很多提议(Janic 2005;Obraczka 1998)。在所有解决方法中，一个主要的问题是为信息传播创建通信通路。在实践中，这里包含了大量管理工作，在很大情况下需要人工干预。而且，只要无集中的提议，ISP是不愿意支持多播的(Doit 等 2000)。

随着点对点技术，尤其是结构化覆盖网络管理的出现，通信通路的创建更简单了。由于点对点解决方法通常是部署在应用层，因此已引入了多种应用层多播技术。在本节中将简要地介绍这些技术。

除了创建显式的通信通路，多播通信还可以以其他方法来完成。本节还将介绍基于gossip的信息传播，它提供了一种简单(但往往效率更低)的多播方法。

4.5.1 应用层多播

应用层多播的基本思想是结点组织成一个覆盖网络，然后用它来传播信息给其成员。一个重要的因素是，网络路由器不在组成员中。因此，覆盖网络的结点之间的连接可以跨越多个物理链接，于是，与网络层路由的消息相比，覆盖网络之中的路由消息并不是最佳的。

一个关键的问题是覆盖网络的构建。这里主要有两种方法(El-Sayed 2003)。第一种是，结点本身直接组织成树，这意味着每对结点之间只有唯一一条路径。另一种方法是，结点组织成一个网状网络，其中的每个结点都有多个邻点，而且一般情况下，每对结点中间有多条路径。这两种方法之间的主要区别是，后者往往能提供更高的健壮性：如果某个连接断开(例如某个结点失效)，无需立即重组整个覆盖网络，仍然可以传播信息。

为了使事情更加具体，我们来看看一个相对简单的方案，即在Chord中构建多播树，见第2章的介绍。这种方案最初是为Scribe(Castro 等 2002)提出的，它是一种应用层多播方法，构建在Pastry(Rowstron 和 Druschel 2001)之上。Pastry也是一种基于DHT的点对点系统。

假设某个结点要开始一个多播会话。它只需生成一个多播标识符(假设为mid)，这只是一个随机选择的160位密钥。然后查找succ(mid)，它负责该密钥的结点，把它提升为多播树的根结点，用来把数据发送给感兴趣的结点。为了加入该树，结点P只需运行LOOKUP(mid)操作，其结果是，含请求加入到多播组mid的查找消息将从P路由到succ(mid)。正如前面已介绍的那样，路由算法将在第5章详细解释。

在成为根结点时，加入请求要通过多个结点。假设它首先到达结点Q。如果Q从没有见过加入到mid的请求，那么它将成为该多播组的一个**转发器**(forwarder)。此时，P成为了Q的一个孩子，且Q继续把加入请求转发到根结点。如果下一个结点(假设为R)仍不是转发器，它将成为一个转发器，记录Q为其孩子，并继续发送加入请求。

另一方面，如果Q(或R)已经是mid的一个转发器，它仍会把前一个发送方(即分别为

P 或 Q)记录为其孩子，但不再需要把加入请求发送给根结点，因为 Q(或 R)已经是多播树的成员了。

按照定义，显式地请求加入到多播树的结点(如 P)同时也是转发器。这种方案的结果是，我们在覆盖网络上构建了含两种类型的结点的多播树：一种是纯粹的转发器，起帮助作用；另一种是兼作转发器的结点，它含有加入树的显式请求。至此，多播很简单：结点只是通过反复运行 LOOKUP(mid)操作，把多播消息发送给树的根结点，这样，消息就可以沿着树被发送。

注意，这里只是简要地描述了 Scribe 中的多播，并没有完全反映其原始设计思想。鼓励感兴趣的读者去详细阅读(Castro 等 2002)。

覆盖网络的构建

从上面简要的描述可以清楚地看出，一旦把结点组织成了一个覆盖网络，构建一棵树并不困难，但要构建一棵高效的树却并不简单。注意，在目前的介绍中，对参与树的结点的选择并没有考虑任何性能问题：它纯粹是基于通过覆盖网络的消息的(逻辑)路由。

为了理解这个问题，来看看图 4.31。该图显示了只有 4 个结点的小集合，它们组织成了一个简单的覆盖网络，其中结点 A 形成了多播树的根结点。穿越物理链接的代价如图所示。只要 A 多播一个消息给其他结点，可以看到，该消息将穿越每个链接〈B, Rb〉、〈Ra, Rb〉、〈Rc, Rd〉和〈D, Rd〉两次。如果我们构建的不是从 B 到 D 而是从 A 到 C 的覆盖链接，那么该覆盖网络就更高效。如果要穿越〈Ra, Rb〉和〈Rc, Rd〉链接，这种配置就可以节省一半。

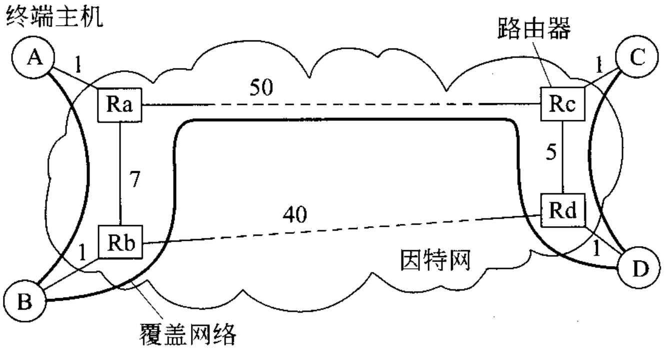


图 4.31 覆盖网络的链接和实际网络层中路由器的链接之间的关系

应用层多播树的质量通常以三种不同的尺度来度量：链接树、相对延时补偿和树成本。**链接树**(link stree)定义每个链接，并计算数据包通过同一链接的次数(Chu 等 2002)。链接树比 1 大，是因为尽管从逻辑上来说，一个数据包可以沿两个不同的连接转发，但这些连接中的一部分实际上对应的是相同的物理链接，如图 4.31 所示。

伸缩(stretch)或**相对延时补偿**(Relative Delay Penalty, RDP)计算覆盖网络中两个结点之间的延时比，以及两个结点在底层网络中的延时。例如，在覆盖网络中，从 B 到 C 的消息经过路由器 B→Rb→Ra→Rc→C，总成本为 59 个单位。但是，在底层网络中，消息可能按路径 B→Rb→Rd→Rc→C 路由，其总成本为 47 个单位，伸缩比为 1.255。显然，在构建覆盖网络时，其目标是最小化这种总的伸缩比，或所有结点对的平均 RDP。

最后,**树成本**(tree cost)是一个全局尺度,往往与最小化总的链接代价有关。例如,如果某个链接的代价就是两个端结点之间延时,那么,优化树代价就相当于找到一棵最小生成树,其中,传播信息到所有结点的总时间最小。

为了简化问题,假设一个多播组具有一个相关的已知结点,它负责跟踪加入到树的结点。当一个新结点发出一个加入请求时,它会与这个**会合结点**(rendezvous node)联系,以获得成员列表(可能是一部分)。其目的是选择可以作为该新结点的父亲的最好成员,它该选择谁?这里有多种可选的不同建议,其解决方法也很不相同。

例如,考虑只有一个源结点的多播组。在这种情况下,最好结点的选择是显而易见的:它应该就是源结点(因为在这种情况下可以保证伸缩比等于1)。但是,在这样做时,要引入一个星形拓扑,其中源结点位于中间。不难想象,源结点很容易过载。换句话说,结点的选择往往限制在这样一种方法:只能选择那些其邻点数不多于k个的结点,其中k为设计参数。这种限制使得树创建算法大大地复杂化了,一个好的解决方法可能要求已有树的一部分进行重构。

(Tan等2003)对这个问题的各种解决方法提供了一个扩展概述和评价。作为说明,让我们来仔细看看一个特殊的类型,称为**交换树**(switch-tree)(Helder和Jamin 2002)。其基本思想很简单。假设我们已有一个多播树,其中唯一的一个源结点作为根。在该树中,结点P通过删除到当前父结点的链接,创建到另一个结点的链接,就可以交换其父结点。交换链接的唯一限制是,新的父结点永远也不可能是以P为根结点的子树的一个成员(因为如果是的话,就会分解这棵树,并生成一个循环),新的父结点也不能有太多的后继子结点。需要后者是为了减少任一结点转发消息的负载。

有不同的标准可用于决定交换父结点。简单的一种是优化到源的路由,当消息要多播时可以有效地最小化延时。每个结点有规律地接收其他结点的信息(后面将解释这样做的一种特定方法)。此时,结点根据从路由器到源的延时,可以评估另一个结点是否更适合作为其父结点,如果是,则开始交换父结点。

另一种标准是,判断到其他潜在的父结点的延时是否比到当前父结点的更短。如果每个结点都以此为标准,那么理论上结果树的总延时为最小的。换句话说,这也就是前面已解释过的优化树成本的一个示例。但是,要构建这样的树需要更多的信息,但事实证明,这种简单的方案对构建近似的最小生成树很有启发性。

作为一个示例,让我们来考虑结点P从其父结点的邻结点接收信息的情况。注意,这些邻结点构成了P的祖父结点以及P的父结点的兄弟结点。结点P可以评估到这些结点的延时,从而选择延时最小的结点(假设是Q)为其新的父结点。结点P发送一个交换请求给Q。为了防止因并发交换请求而产生的循环,如果结点已有一个未完成的交换请求,那么它将拒绝处理其他所有的进入请求。这就导致这样一种情况:只有完全无关的交换请求才能同时执行。而且,P为Q提供了足够的信息,使得Q可以知道两个结点有相同的父结点,或者Q是祖父结点。

这里还没有解决的一个重要问题是结点的失效。在交换树情况下,一个简单的解决方法是,只要某个结点知道其父结点失效了,那么它就以根结点作为其父结点。此时,就可以像正常情况一样进行优化处理,最终把该结点放置在多播树的恰当位置。(Helder和Jamin 2002)介绍的经验表明,这种结果树的确接近于最小生成树。

4.5.2 基于 gossip 的数据通信

对传播信息来说，一种日益重要的技术是依靠感染行为(epidemic behavior)。研究人员观察了疾病是如何在人群中扩散的，于是进行了很长时间的调查，看看是否可以开发些简单的技术来在超大型分布式系统中进行信息扩散。这些**感染协议**(epidemic protocol)的主要目的是使用本地信息在大型结点集中快速地传播信息。

要解释这些算法的一般原理，我们假定特定数据项的更新都是从单个结点发起的。这些就可以避免写冲突。下面内容就是基于有关流传算法的经典论文(Demers 等 1987)。有关流传信息传播的最近介绍可参见(Eugster 等 2004)。

4.5.2.1 信息传播模型

正如其名所表示的那样，感染算法是基于流行病理论的。流行病研究的是传染病的传播的。在大型分布式系统中，要传播的不是疾病，而是信息。对分布式系统中有关传染病的研究的目标也是完全不同的：身体组织是尽最大努力防止传染病在大量人群中传播，而分布式系统的感染算法设计人员则试图尽可能快地用新信息“感染”所有结点。

借用来自传染病的术语，分布式系统的某个结点如果具有要传播到其他结点的数据，则称之为**已感染的**(infected)。还没有接收到数据的结点称为**易受感染的**(susceptible)。最后，不会传播其数据的已更新结点称为**已隔离的**(removed)。这里我们假定可以区分新旧数据，因为其数据可能已加时间戳或版本号了。

一种流行的传播模型是 **anti-entropy**。在这种模型中，结点 P 随机选取另一结点 Q，然后与 Q 交换更新信息。这里，交换更新信息的方法有如下三种：

(1) P 只是把它自己的更新信息发出给 Q，即为基于 push 的方法；

(2) P 只是从 Q 那里获得更新信息，即为基于 pull 的方法；

(3) P 和 Q 相互发送更新信息给对方，即为基于 push-pull 的方法。

如果要快速传播更新信息，那么只是使用基于 push 的方法是一个不好的选择。显然，这可以如下来理解。首先，注意到，在基于 push 的方法中，更新信息只能由已感染的结点来传播。但是，如果已感染的结点很多，那么每个结点选择一个易感染的结点的概率就相对小。同样，某个结点要长时间保持为易感染的机会也简单，因为它不会被已感染的结点选中。

相反，在基于 push 的方法中，如果易感染的结点很多，那么其工作得就更好。在这种情况下，传播更新信息基本上是由易感染的结点来触发的。此时，这种结点接触已感染的结点的机会也就大，从而可以从那里获得更新信息，并同样变成为已感染的结点。

这表明，如果只有一个结点是已感染的，使用 anti-entropy 模型的任意一种方法，更新信息都会通过所有结点迅速传播，但 push-pull 方法最好(Jelasity 等 2005a)。在一段时间里，如果每个结点作为发起者与随机选择的另一个结点进行了至少一次更新信息交换，则定义为一**轮**(round)。可以证明，传播单个更新信息到所有结点需要 O(log(N))轮，其中 N 为系统的结点数。这的确表明，传播更新信息很快，而且是可扩展的。

这种方法的一个特殊变体称为**流言传播**(rumor spreading)，或简称为 gossiping。它的

工作原理如下。如果结点P正好刚更新数据项x,那么它将与任意结点Q通信,并把更新信息发送给Q。但是,有可能Q被另外一个结点已更新了。在这种情况下,P可能不再传播该更新信息的概率为1/k。换句话说,它变为了已隔离的。

gossiping完全类似于现实生活。当Bob有热点新闻要传播时,他可能打电话给他的朋友Alice。像Bob一样,Alice也可能告诉她的朋友。但是,当她给她的某个朋友(比如是Chuck)打电话时,Chuck可能会告诉她,他已经知道这个新闻了。Alice可能就不再打电话告诉其他朋友了,如果他们已经知道了,该如何是好呢?

事实证明,gossiping是快速传播新闻的很好方法。但是,它不能保证所有结点都真的已更新了(Demers等1987)。可以证明,当有大量结点时,保持未更新的(即仍为易感染的)结点的系数s满足如下等式:

$$s = e^{-(k+1)(1-s)}$$

图4.32显示了k的函数ln(s)。例如,如果k=4,ln(s)=-4.97,因此s小于0.007,表明只有不到0.7%的结点仍为易感染的。然而,仍需要特殊的方法来保证这些结点也被更新。将anti-entropy与gossiping组合起来就可获得成功。

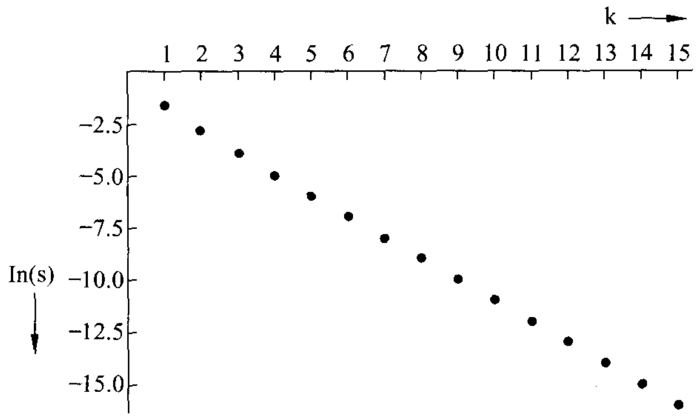


图4.32 在gossiping中,未更新结点的系数s与参数k的关系。该图显示了k的函数ln(s)

感染算法的一大优点是其可扩展性,因为,与其他传播方法相比,在这种方法中,其进程间的同步数目相对要小。对广域系统来说,(Lin和Marzullo 1999)显示,把实际的网络拓扑结构因素考虑进去可以获得更好的结果。在该文章的方法中,只与少量其他结点相连接的结点,其进行通信的概率相对高些。这里的一个基本假设是,这些结点构成了一个到网络的其他远程部分的桥,因此,它们可以很快地进行通信。这种方法又称为**有向gossiping**(directional gossiping),可以衍生出不同的变体。

这个问题涉及到大多数感染解决方法所做的一个重要假设,即某个结点可以随机选取要通信的其他结点。这意味着,原则上,所有结点之间都是相连的。在大型系统中,这种假设是永远做不到的。

幸运的是,实际上并不需要这样。正如第2章介绍的那样,维护一个可持续更新的部分视图,就可以把结点集组织成一个随机图。通过定期地更新每个结点的部分视图,随机选择就不再是问题了。

4.5.2.2 删除数据

感染算法特别擅长传播更新信息。但是，它们有一个很奇怪的副作用：传播数据项的删除很困难。这个问题的实质在于，数据项的删除破坏了该项的所有信息。因此，当某个数据项从结点删除时，该结点最终将接收该数据项的旧副本，并把这些看作是它以前从没有过的更新信息。

解决这个问题的技巧是把数据项的删除记录为另一次更新，并保持这个删除记录。这样，旧副本就不会被认为是新内容了，而只是看作为由删除操作已更新的版本。删除记录是通过传播**死亡证书**(death certificate)来完成的。

当然，死亡证书带来的问题是，最后必须清除掉它们，否则，每个结点都将逐渐构建一个巨大的本地数据库，其中存储的是已删除的数据项的历史信息。(Demers 等 1987)建议使用休眠死亡证书。每个死亡证书在创建是加上时间戳。如果可以假定在已知的有限时间里，更新信息可以传播给所有结点，那么在最大传播时间后，就可以删除死亡证书。

但是，为了确保删除信息真的传播给了所有结点，只需很少的一些结点维护休眠死亡证书，永远别删除掉。假设结点 P 为数据项 x 维护了这样一个证书。如果 x 的一个废旧更新信息到达了结点 P，那么 P 所要做的只是再次传播 x 的死亡证书即可。

4.5.2.3 应用

为了结束本内容的介绍，让我们来看一些感染算法的有趣应用。我们已经介绍了传播更新信息，这可能是部署最广泛的应用了。同样，在第 2 章中，我们介绍了提供结点的定位信息可以帮助构建特殊的拓扑结构。同样，gossiping 也可用于发现具有广域输出链接的结点，从而可应用前面介绍的有向 gossiping。

另一个有趣的应用领域是收集或汇集信息(Jelasity 等 2005b)。考虑如下的信息交换。开始时，每个结点 i 选择一个任意数，例如 x_i。当结点 i 与结点 j 通信时，它们相互如下更新其值：

$$x_i, x_j \leftarrow (x_i + x_j) / 2$$

显然，经过交换后，i 和 j 具有相同的值。事实上，不难看出，最后所有结点都具有相同的值，即所有初始值的平均值。传播速度是指数性的。

使用什么来计算这个平均值呢？考虑这样一种情况：对所有结点 i，x_i 设置为 0，x_1 除外，x_1 设置为 1：

$$x_i = \begin{cases} 1, & i=1 \\ 0, & i>1 \end{cases}$$

如果有 N 个结点，那么最后每个结点都将计算这个平均值，该值为 1/N。因此，由于每个结点 i 为 x_i，可以估计出系统的大小。这个信息可用来动态地调整各种系统参数。例如，部分视图的大小(即每个结点保持联系的邻点数目)应与总结点的数目有关。知道了这个数，就使得结点可以动态地调整其部分视图的大小。的确，这可以视作为自我管理的一种属性。

当结点有规律地加入或离开系统时，计算平均值很困难。这个问题的一个实用解决方法是引入“时间点”的概念。假设结点 1 是稳定的，它只是偶尔开始一个新的时间点。当结

点 i 第一次见到一个新的时间点时，将自己的变量 x_i 设置为 0，重新计算平均值。

当然，也可以计算其他结果。例如，可以不是从某个固定结点（x_1）开始平均值的计算，而是以如下方式选取一个随机结点。开始时，每个结点 i 把 x_i 设置为相同数段（如[0，1]）中的某个随机数，并把它持久保存为 m_i。在结点 i 和 j 之间进行交换时，将其值改变为：

$$x_i, x_j \leftarrow \max(x_i, x_j)$$

对每个 $m_i < x_i$ 的结点 i，都不能作为平均值计算的开始者。最后，只有一个结点胜出。当然，要知道某个结点不能胜出很容易，但要知道某个结点可以胜出则要困难得多，因为不知道是否所有的结果都进行了比较。这个问题的解决方法还是挺乐观的：每个结点总是假设自己是胜出者，直到被否定为止。此时，它只需把用于计算平均值的变量设置为 0 即可。注意，到现在为止，可能并行运行多种不同的计算方法（在我们的示例中，同时进行了最大值和平均值的计算）。

本章小结

强大灵活的进程间通信功能是任何分布式系统中必需的。在传统的网络应用中，通信常常是通过基于传输层提供的低层消息来传递原语的。中间件系统要解决的一个重要问题是：提供更高层次的抽象，与传输层接口所提供的支持相比，这能更加方便地表述进程间的通信。

使用最为广泛的抽象方法之一就是远程过程调用（remote procedure call，RPC）。RPC 的本质是一种通过过程来实现的服务，该过程的主体在服务器上执行。只向客户提供过程的特征，也就是该过程的名字以及参数。当客户调用该过程的时候，其客户端实现——称为客户存根（stub），将会负责将参数值封装进消息中，然后将消息发送给服务器。由服务器来调用实际过程并在消息中返回结果，由客户存根从返回的消息中提取结果值，然后将结果值传递回调用它的应用程序。

RPC 提供了同步通信功能，通过这些功能，可以将客户阻塞，直到服务器送回应答为止。虽然这种机制的某些变体放松了模型中对同步的要求，但是实践证明，通用的高级面向消息模型会更加方便。

在面向消息的模型中，存在的问题是，通信是否是持久的，以及通信是否是同步的。持久通信的本质是，提交进行传输的消息由通信系统来存储，直到系统将其传输给接收方为止。换句话说，发送方和接收方都不必保持运行来等待消息传输。而在瞬时通信中，不提供存储功能，因此发送消息时接收方必须等待接收该消息。

在异步通信中，允许发送方在将消息提交并进行传输后立即继续运行——甚至可以在消息实际发送出去之前就继续运行。而在同步通信中，发送方会被阻塞，至少要到消息得到接收之后才能释放。发送方还有可能要等到消息已经交付给接收方之后甚至等到接收方进行响应之后才能继续运行，最后一种情况就像 RPC 一样。

面向消息的中间件模型一般提供持久异步通信，用在 RPC 不适用的场合。它们主要用来协助将高度分散的数据库集成进大规模信息系统中。其他的应用还包括电子邮件和工作流。

流是一种完全不同的通信方式，其主要问题是两个连续的消息是否有时间上的联系。在连续数据流中，每个消息都规定了端到端的最大延迟时间。另外，发送的消息还要受端到

端最小延迟时间的约束。这种连续数据流的典型例子包括视频流和音频流。这种时间上的联系以及希望底层通信子系统提供的服务质量往往是难于说明和实现的。使得事情变得复杂的因素之一是信号的延迟不稳定性。即使平均性能是可接受的，传输时的实际延迟时间的变化可能会导致性能恶化到不可接受的程度。

最后，分布式系统中一个重要的通信协议是多播。其基本思想是从发送方到多个接收方进行信息传播。这里讨论了两种不同的方法。首先，通过创建从发送方到接收方的树，可以实现多播。如果明白了结点是如何自组织为点对点系统的，解决方法也就很显然，以非集中的方式动态地创建树。

信息传播的另一个重要解决方法是，使用感染协议。事实证明，这种协议很简单，仍然却很健壮。除了进行简单的传播消息，感染协议还可以有效地在大型分布式系统中进行信息的汇集。

习　题

1. 在很多分层协议中，每一层都有自己的报头。如果每个消息前部只有单个报头，其中包含了所有的控制信息，无疑会比使用单独的多个报头具有更高的效率。为什么不这么做？
2. 为什么传输层通信服务往往不适于构建分布式应用程序？
3. 一种可靠的多播服务允许发送方向一组接收方可靠地传递消息。这种服务是属于中间件层还是属于更低层的一部分？
4. 考虑一个含有两个整型参数的过程 incr。该过程将两个参数的值分别增加 1。现在假定调用它时，使用的两个参数是同一个变量，比如 incr(i, i)。如果 i 的初始值是 0，在执行引用调用之后 i 将变为什么值？如果使用复制/还原调用呢？
5. C 语言中有一种称为联合(union)的结构，其中记录(在 C 语言中称为结构)的字段可以用来保存几种可能值中的一个。在运行时，没有可靠的办法来分辨其中保存的是哪一个值。C 的这种特性是否与远程过程调用有某些相似之处？请说明理由。
6. 处理 RPC 系统中参数转换的一种方法是，每台机器以自己系统使用的表示方式来发送参数，由另一方在必要的情况下进行转换。可以通过首字节中的代码来表示发送消息机器所用的系统。然而，由于要在首个字中找到开头的字节这本身也是一个问题，这种方法能行得通吗？
7. 假定客户通过异步 RPC 对服务器进行调用，随后等待服务器使用另一异步 RPC 返回结果。这种方法与客户执行常规的 RPC 有没有什么不同？如果使用的是同步 RPC 而不是异步 RPC，情况又如何呢？
8. 在 DCE 中，服务器在守护程序中注册自身。如果换一种方法，也可以总是为它分配一个固定的端点，然后在指向服务器地址空间中对象的引用中就可以使用该端点。这种方法的缺陷在哪里？
9. 给出一种用来让客户绑定到瞬时远程对象的对象引用的实现实例。
10. 请描述一下在客户和服务器进程间使用套接字时如何进行无连接通信。
11. 请解释 MPI 中 mp_bsend 原语和 mp_isend 原语之间的区别。
12. 假定只能使用瞬时异步通信原语，再加上异步 receive 原语，如何实现用于瞬时同步通

信的原语？

13. 假定只能够使用瞬时同步通信原语，如何实现用于瞬时异步通信的原语？
14. 通过 RPC 实现持久异步通信有意义吗？
15. 在本章中我们讲过，为了自动启动一个进程以从输入队列中获取消息，常常要使用守护程序来监视输入队列。请给出一种不使用守护程序的实现方法。
16. 在 IBM WebSphere 以及许多其他消息队列系统中，路由表是手工配置的。描述一种自动完成配置工作的简单方法。
17. 在持久通信中，接收方一般拥有自己的本地缓冲区，如果接收方不在运行状态，可以将消息放入该缓冲区中去。为了创建这种缓冲区，必须指定它的大小。分成两方进行辩论：一方认为这种指定缓冲区大小的行为是可取的，而另一方反对这种指定大小的行为。
18. 请说明为什么瞬时同步通信在可扩展性方面存在固有的问题，以及如何解决这些问题。
19. 给出一个将多播应用于离散数据流的例子。
20. 假设在一个传感器网络中，传感器没有给已测出的温度添加时间戳，而是立即发送给操作者。这是否足以保证最大端到端延迟时间？
21. 当一组计算机组成一个（逻辑上或者物理上的）环时，如何确保传输延迟不超过允许的最大端到端延迟时间？
22. 当一组计算机组成一个（逻辑上或者物理上的）环时，如何确保传输延迟不小于允许的最小端到端延迟时间？
23. 尽管多播在技术上是可行的，但在因特网中对它的部署支持却很少。这个问题的答案可以在实际的商业模型中找到：没有人真的明白如何利用多播来获利。你能创造什么方案吗？
24. 通常，应用层多播树可以用关系图来优化，其中，关系图是根据延时或跳数来度量的。请给出一个示例，说明这种度量方法会导致很糟糕的树。
25. 当在非结构化的点对点系统中查找文件时，限制查找与你有类似文件的结点可能会有用。请解释如何 gossiping 能有助于找到这些结点。

第5章

命名系统

名称在所有计算机系统中都起着重要的作用。它们用来共享资源、唯一标识实体、指向位置等。命名的一个重要结果是可以把名称解析为它所指向的实体。因而，名称解析允许进程访问命名的实体。要解析名称，需要实现命名系统。分布式系统中的命名与非分布式系统中的命名之间的区别在于命名系统实现的方式。

在分布式系统中，命名系统的实现本身通常是分布在多台计算机上的。该分布是如何完成对命名系统的效率和可扩展性起着关键的作用。在本章中，我们集中讨论在分布式系统中使用名称的三种不同的重要方法。

首先，在讨论一些与命名有关的常见问题之后，我们将进一步研究用户友好的名称的组织和实现。这样的名称的典型例子包括用于文件系统和万维网的名称。建立世界范围的、可扩展的命名系统是这类名称主要关注的问题。

其次，名称用来以与实体当前位置无关的方式来定位它们。实践表明，用户友好的命名系统并不特别适合支持大量移动实体。大多数名称甚至与实体的位置毫不相干。还需要其他的命名机制，例如，用于移动电话和分布式散列表的命名机制。在移动电话命名机制中，名称是与位置无关的标识符。

最后，人们往往更愿意利用不同的字符来描述实体，这就需要把属性描述解析为与该描述相关的实体。这种名称解析类型非常困难，我们将给予特别关注。

5.1 名称、标识符和地址

我们首先来仔细看看名称实际上是什么。在分布式系统中，名称是由位或字符组成的字符串，用来指向一个实体。分布式系统的实体几乎可以是任何事物。典型的示例包括如主机、打印机、磁盘和文件等资源。通常得到明确命名的众所周知的其他实体示例还有进程、用户、邮箱、新闻组、Web页面、图形窗口、消息和网络连接等。

对实体可以进行操作。例如，像打印机这样的资源可以提供一个界面，在打印机的接口包含打印文档、查询打印工作的状态等操作。此外，像网络连接这样的实体可以提供发送和接收数据、设置服务质量参数和查询状态等操作。

如果要对实体进行操作，就需要访问实体，因此需要一个**访问点**(access point)。在分布式系统中，访问点是另一种特殊类型的实体。访问点的名称称为**地址**(address)。一个实

体的访问点地址也可简称为该实体的地址。

一个实体可以提供多个访问点。打个比方，电话号码可以视为一个人的访问点，其中电话号码相当于地址。实际上，如今许多人都拥有多个电话号码，每个号码都相当于是可以联系到他们的一个访问点。在分布式系统中，访问点的一个典型示例就是运行某个特定服务器的主机，可以通过IP地址和端口号（即服务器的传输层地址）的组合来构成其地址。

一个实体经过一定的时间后可能改变其访问点。例如，当一台移动式计算机移动到另一个地方时，通常会为它分配一个与以前不同的IP地址。同样，一个人到达另一个城市或国家后，通常也需要改变电话号码。与此类似，变换工作或者改变因特网服务提供商（ISP）都意味着要改变电子邮件的地址。

因此，地址就是一种特殊类型的名称：它指向实体的访问点。因为访问点是与实体紧密关联的，所以使用访问点的地址作为相关实体的常规名称似乎是非常方便的。但是，在实际中却很少这样做，因为这种命名往往很不灵活而且与用户不友好。

例如，定期地重新组织一个分布式系统是很常见的工作，使得一台特定服务器运行在与以前不同的主机上。而该服务器以前所在的计算机可能会被重新指定一个完全不同的服务器。换句话说，一个实体可以很容易改变访问点，而一个访问点也可能会重新分配给不同的实体。如果用地址来指向实体，那么由于访问点发生改变或者重新分配给了另一个实体，就可能得到一个无效的引用。因此，更好的办法是，某个服务与独立的名称相关，而该名称与相应的服务器的地址无关。

同样，如果一个实体提供了多个访问点，那么哪个地址用作引用是不清楚的。例如，许多公司将其Web服务分布在几台服务器上。如果用这些服务器的地址来作为Web服务的引用，那么哪个地址应该作为最佳选择是不明显的。同样，更好的解决方法是让Web服务具有一个与不同Web服务器的地址无关的单一名称。

这些实例说明，一个独立于实体地址的名称通常是比较简单的，而且使用更为灵活。我们称这样的名称是与**位置无关**（location independent）的。

除了地址之外，还有其他类型的名称值得特别注意，例如，用来唯一标识实体的名称。真正的**标识符**（identifier）是具有以下属性的名称（Wieringa 和 de Jonge 1995）：

（1）一个标识符最多引用一个实体。

（2）每个实体最多由一个标识符引用。

（3）一个标识符始终引用同一个实体（也就是说，标识符永远不会重新使用）。

通过使用标识符，使得明确地引用某个实体变得更加容易。例如，假设两个进程都通过标识符引用了一个实体。如果要检查两个进程引用的是否是同一实体，那么只需检验两个标识符是否相等就可以了。如果这两个进程使用的是常规的、非唯一性的、非标识性名称，那么仅做这样的检验是不够的。例如，名字John Smith不能作为某个人的唯一引用。

同样，如果一个地址可以重新分配给不同的实体，那么就不能使用地址作为标识符。考虑电话号码的使用，一个号码通常指向的是同一个人或公司，这是相当稳定的。但是，不能使用电话号码作为一个标识符，因为经过一段时间电话号码可能会重新分配。因此，Bob的新面包店可能会长期接到找Alice的旧五金商店的电话。在这种情况下，最好让Alice使用一个真正的标识符，而不是使用她的电话号码。

地址和标识符是两个重要的名称类型，每个类型用于不同的目的。在许多计算机系统

中，地址和标识符仅使用计算机可读的形式（即位串的形式）来表示。例如，以太网地址就是一个任意的48位字符串。同样，内存地址通常用32位或64位字符串来表示。

另一种重要的名称类型是为用户使用而定制的，也称为**用户友好的名称**（human-friendly name）。相对于地址和标识符而言，用户友好的名称通常用字符串来表示。这些名称表现为各种各样的形式。例如，UNIX系统中的文件，最多可具有255个字符的字符串名称，它完全是由用户来定义的。同样，DNS名称是用相对简单的不区分大小写的字符串来表示的。

有了名称、标识符和地址后，我们就可以进入本章的主题：如何把名称和标识符解析为地址？在介绍不同的解决方法之前，认识到分布式系统中的名称解析与消息路由往往有紧密关系是很重要的。原则上，命名系统含有一个**名称到地址的绑定**（name-to-address binding），其最简单的形式只是一个（name, address）对的表。但是，在分布式系统中，要跨越多个大型网络以及需要为很多的资源命名，一个集中式的表无法满足工作。

通常的做法是，把名称分解成诸如ftp.cs.vu.nl的多个部分，通过迭代地查询这些部分来解析该名称。例如，某个客户需要知道名为ftp.cs.vu.nl的FTP服务器的地址，那么它首先解析nl，找到服务器NS(nl)对应的以nl结尾的名称，然后把名称的其余部分传递给服务器NS(nl)。该服务器再把名称vu解析为服务器NS(vu.nl)对应的以vu.nl结尾的名称。服务器NS(vu.nl)可以进一步处理剩余的名称ftp.cs。最后就可以得到如下的名称解析请求路由：

NS(.)（NS(nl)≤NS(vu.nl)，ftp.cs.vu.nl的地址

其中，NS(.)表示的是可以返回NS(nl)地址的服务器，又称为**根服务器**（root server）。NS(vu.nl)可以返回FTP服务器的真实地址。要注意的是，名称解析与消息路由之间的界线已开始慢慢地变得模糊了。

在后面的章节中，我们将介绍三种不同的命名系统。首先，我们将看看标识符是如何解析为地址。在这种情况下，我们还将看到一个示例，其中的名称解析与消息路由之间真的很难区分。然后，我们将介绍用户友好的名称和描述性名称（即实体是由名称集描述的）。

5.2 无层次命名

前面介绍了标识符可以很方便地唯一表示实体。在很多情况下，标识符只是随机的位字符串，我们可以很方便地用作非结构化或无层次的名称。这种名称的一个重要属性是，它不包含任何有关如何定位其相关实体的访问点的信息。接下来，我们将介绍如何解析无层次名称，或者说，当只给实体的标识符时，如何定位它。

5.2.1 简单方法

首先来考虑两个用于实体定位的简单方法。这两种方法都只适用于局域网。不过，在这种环境中，它们通常能够很好地工作，这使得它们的简便性特别吸引人。

5.2.1.1 广播和多播

设想这样一个分布式系统：它创建于一个能够提供高效广播功能的计算机网络之中。通常情况下，这种功能是由局域网提供的，在这样的网络中，所有的机器都连接在一根电缆上。同样，局域无线网络也属于这一类。

在这种环境中定位实体是一件简单的事情：包含该实体所用标识符的消息会广播到每台机器上，并且请求每台机器查看它是否拥有该实体。只有能够为该实体提供访问点的机器才会发送回复消息，回复消息中包含访问点的地址。

在因特网**地址解析协议**（Address Resolution Protocol，ARP）中，这个原理用于在仅仅指定 IP 地址的情况下查找机器的数据链路地址。实质上，机器会在本地网络中广播一个数据包，以查找拥有指定 IP 地址的机器。当消息到达一台机器以后，接收者会查看一下自己的 IP 地址是否与监听到的被请求的 IP 地址相同。如果是，那么它会发送一个回复数据包，其中包含它的以太网地址。

随着网络的膨胀，广播开始变得低效。不仅网络带宽被请求数据包浪费，而且更为严重的是，可能有太多的主机被它们不能回答的请求中断。一种可能的解决方法是转换成多播，通过使用多播，只有符合条件的一组主机才会接收到请求。例如，以太网直接在硬件上支持数据链路层多播。

也可以在点到点网络中使用多播来定位实体。例如，通过允许主机加入特定的多播组，因特网就可以支持网络层多播。这样的组由多播地址来标识。当主机向一个多播地址发送消息时，网络层会提供最尽力的服务来把消息发送给全部组成员。（Deering 和 Cheriton 1990；Deering 等 1996）讨论了多播在因特网中的高效实现。

多播地址可以用作对多播实体的通用定位服务。例如，设想这样一个组织：组织中的每个员工都拥有自己的移动计算机。当这样的计算机连接到本地可用的网络中时，网络会动态地把一个 IP 地址分配给它。另外，它还将加入一个特定的多播组。当一个进程需要查找计算机 A 时，它会向多播组发送一个“A 在哪里？”的请求。如果 A 连接到了网络上，那么它就会用它当前的 IP 地址进行响应。

使用多播地址的另一种方式就是让它与一个复制的实体相关联，并且使用多播查找最近的实体副本。向该多播地址发送请求时，每个复制实体都会用它当前的（通常的）IP 地址进行响应。选择最近的实体副本的一种粗糙方法就是选择最先回复的实体副本。后面将介绍了更多精确的方法。实践证明，选择最近的实体副本通常没那么容易。

5.2.1.2 转发指针

另一种用于移动实体定位的流行方法是使用转发指针（Fowler 1985）。它的原理很简单：当实体从 A 移动到 B 时，它将在后面留下一个指针，这个指针指向它在 B 中的新位置。这种方法的主要优点是它很简便：一旦找到实体以后（比如使用传统的命名服务），客户就可以顺着转发指针形成的链来查找实体的当前地址。

转发指针也存在许多重要的缺点。首先，如果不采取特殊措施，那么链可能会特别长，以致定位实体的开销会变得很大。其次，只要需要，链中的所有中间位置就必须维护它们的那一部分转发指针链。第三，是它所在链很脆弱，易断开。不管是什么原因，只要有一个转

发指针丢失，就无法再到达实体。因此，一个重要的事情就是让指针链相对短一些，并且确保转发指针是健壮的。

为了更好地理解转发指针的工作方式，让我们来考虑一下它们与分布式对象一起使用的情况。按照 **SSP 链**(SSP chain)(Shapiro 等 1992)中的方法，每个转发指针都以(客户存根，服务器存根)对的形式实现，如图 5.1 所示。(注意，在 Shapiro 最初的术语中，服务器存根称为子孙，于是有(stub，scion)值对，即(存根，后裔)对。)服务器存根要么包含实际对象的本地引用，要么包含该对象对远程客户存根的本地引用。

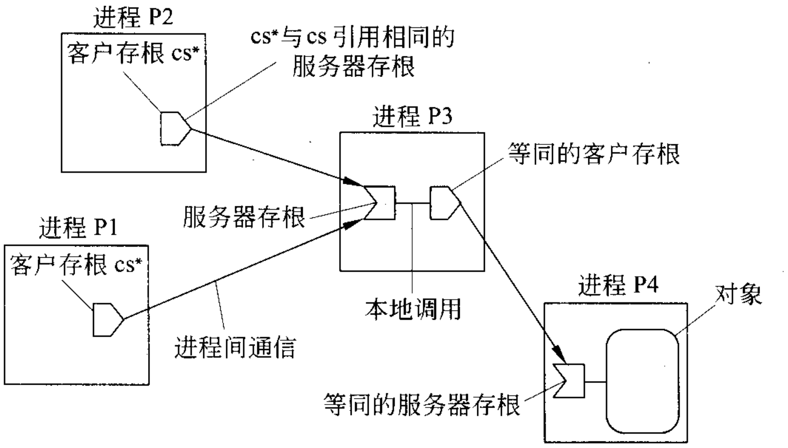


图 5.1　使用(客户存根，服务器存根)对的转发指针的原理

当对象从地址空间 A 移到地址空间 B 时，它会把一个客户存根留在 A 中，并且在 B 中安装一个引用它的服务器存根。这种方法的一个有趣之处是：移动细节对客户是完全透明的。客户唯一能看到对象的就是一个客户存根。客户存根如何转发调用的方法，以及把调用转发到了什么位置，客户无法看到。还要注意的是，这种转发指针的使用与查找地址是不同的。客户请求是沿着链转发到实际对象的。

为了简化(客户存根，服务器存根)对的链，对象调用会携带一个客户存根的标识，这是调用发起的地方。这个客户存根的标识由客户的传输层地址加上一个本地产生的、用于标识该存根的号码组成。当调用到达位于当前位置的对象以后，会向发起调用的客户存根发送回一个响应。当前位置中包含了对象的当前位置，客户存根会把自己的对应服务器存根调整为对象当前位置中的那一个服务器存根。图 5.2 显示了该原理。

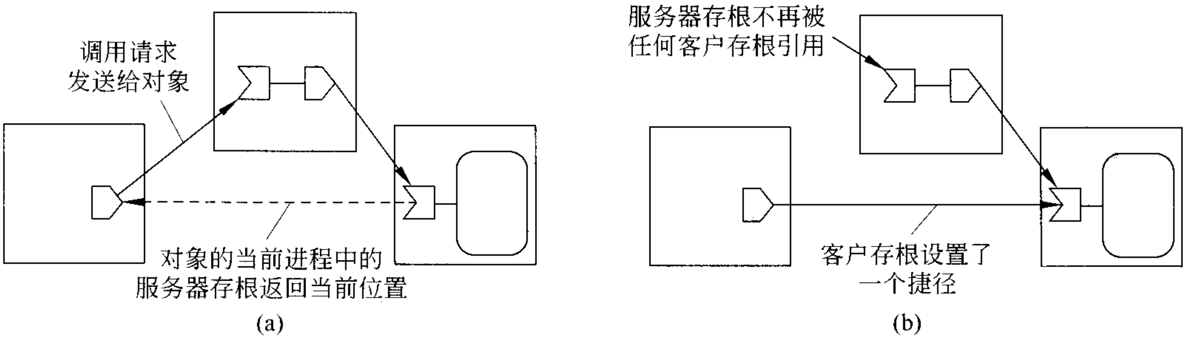


图 5.2　通过在客户存根中存储一个捷径来重定向转发指针

直接向起始客户存根发送响应或沿着转发指针的相反路线发送响应，二者各有所长。

对于前者来说，通信要更快一些，这是因为需要经过的处理较少。另一方面，使用前者时只有起始客户存根可以调整，而后者可以调整所有中间存根。

当服务器存根不再被任何客户引用时，就可以删除它。这很像分布式垃圾收集器。这里不值得去讨论这个问题。感兴趣的读者可以参考(Abdullanhi 和 Ringwood 1998; Plainfosse 和 Shapiro 1995; Veiga 和 Ferreira 2005)。

假设图5.1所示的进程P1要把它对对象O的引用传递给进程P2。引用传递的实现方法是在进程P2的地址空间中安装客户存根p的一个副本p′。由于客户存根p′和p引用的是相同的服务器存根，所以转发调用机制和以前一样工作。

如果由(客户存根，服务器存根)对组成的链中有进程崩溃，或者因其他原因而无法到达，那么将会出现问题。可以有几种解决方法。其中一种在Emerald(Jul 等 1988)和LII system(Black 和 Artsy 1990)里面介绍过，这种方法是让创建对象的机器(称为该对象的**宿主位置**(home location))始终保留一个对对象的当前位置的引用。这个引用以一种容错方式进行存储和维护。当链断开时，可以向对象的宿主位置提出请求，询问其对象当前所在的位置。为了允许对象的宿主位置发生变化，可以使用传统的命名服务来记录当前的宿主位置。下面讨论这种基于宿主位置的定位方法。

5.2.2 基于宿主位置的方法

广播和转发指针的使用带来了可扩展性的问题。在大型网络中，过长的转发指针链会导致性能问题，并且容易受到链断开的影响，很难有效地实现广播和多播。

在大型网络中支持移动实体定位是引入**宿主位置**(home location)的方法。这种方法持续跟踪实体的当前位置，可以使用特殊的技术来预防网络故障或进程失效。在实践中，通常选择创建实体的位置作为其宿主位置。

正如前面讨论的那样，对于基于转发指针的定位服务来说，基于宿主位置的方法用来作为一种回退机制。Mobile IP(Johnson 等 2004)介绍了一个采用基于宿主位置的方法，我们在第3章中进行了简要介绍。每个移动主机都使用一个固定的IP地址，所有与该IP地址进行的通信一开始都被转发到移动主机的**宿主代理**(home agent)中。宿主代理位于局域网中，与包含在移动主机IP地址中的网络地址相对应。当一台移动主机转移到另一个网络中时，它会请求一个可以用来通信的临时地址。这种**转交地址**(care-of address)要在宿主代理中注册。

当宿主代理接收到发给移动主机的数据包时，它会查找主机的当前位置。如果主机是在当前本地网络中，那么就只需转发数据包。否则，它会建立一条通往主机当前位置的通道，也就是说，它会把数据组装成IP包，然后发送给转交地址。同时，将把主机的当前位置告诉数据包的发送者。图5.3显示了该原理。注意，IP地址被有效地用作移动主机的标识符。

图5.3还说明了在大型网络中基于宿主位置的方法存在另一个缺点。为了与移动实体通信，客户首先必须与宿主位置进行联系，而宿主位置可能与实体本身处于完全不同的位置。结果是增加了通信延迟。

基于宿主位置方法的另一个重要缺点是它使用了固定的宿主位置。一方面，必须保证

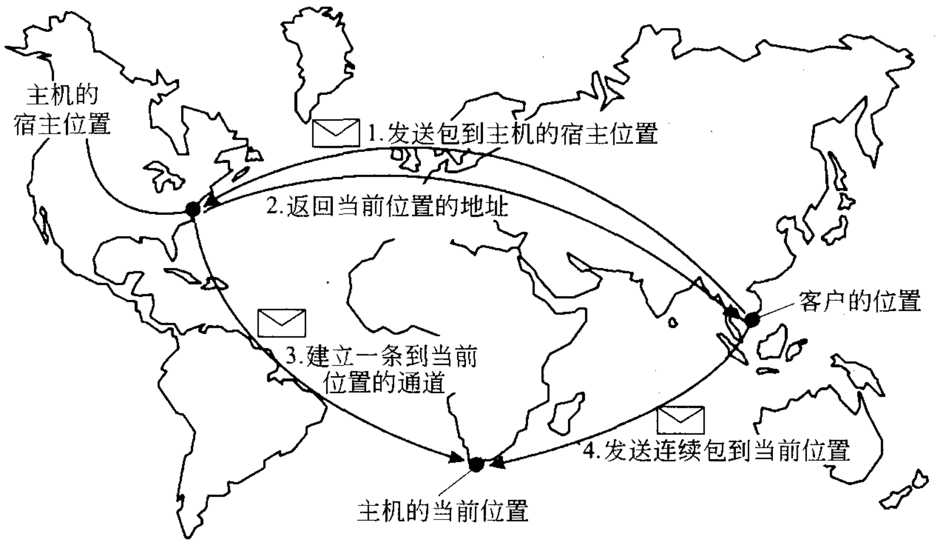


图 5.3 移动 IP 原理

宿主位置始终存在。否则，将无法与实体联系。如果决定把一个长期存在的实体永久转移到网络的一个完全不同的部分中，而不是转移到宿主位置所在的网络部分，那么问题会进一步恶化。在这种情况下，如果能让宿主位置随着主机一起转移就好了。

针对这个问题的一种解决方法是：在传统的命名服务中注册宿主位置，然后让客户首先查找宿主位置所在的位置。由于可以假定宿主位置是相对稳定的，所以在查找到它以后可以有效地缓存它。

5.2.3 分布式散列表

(Balakrishnan 等 2003)简要介绍了各种基于 DHT 的系统。尽管有些很重要的区别影响着这些系统的维护复杂性和查找协议，但 Chord 系统(Stoica 等 2003)很具有代表性。正如第 2 章简要介绍的那样，Chord 系统使用一个 m 位的标识符空间，把随机选择的标识符赋给结点，并把键值赋给特定实体。实体可以是虚拟的任意东西，如文件、进程等。根据所用的散列函数，位数 m 通常为 128 或 160。含有键值 k 的实体位于含有最小标识符 id≥k 的结点之内。该结点称为 k 的后继者，用 succ(k)表示。

基于 DHT 系统的主要问题是要高效地把键值 k 解析为 succ(k)的地址。一个无扩展性的方法是，让每个结点 p 跟踪后继者 succ(p+1)和前继者 pred(p)。在这种情况下，只要结点 p 接收到一个将键值 k 解析的请求，除非 pred(p)<k≤p，否则，它只需将该请求转发给两个邻结点(总有一个是适合的)即可。如果 pred(p)<k≤p，那么结点 p 应把它自己的地址返回给发起请求解析键值 k 的进程。

每个 Chord 结点不是线性地进行键值查找，而是维护一个最多有 m 个实体的**指状表**(finger table)。如果用 FT_p表示结点 p 的指状表，那么有

$$FT_p[i]=succ\ (p+2^{i-1})$$

换句话说，第 i 个实体指向 p 后 2^{i-1}的第一个结点。注意，这些引用实际上是 i 个标识符空间表示的已有结点的快捷键，其中，随着指状表的索引的增加，从结点 p 的快捷距离呈

指数增加。要查找键值 k，结点 p 立即把该请求转发给在 p 的指状表中索引为 j 的结点 q，其中：

$$q = FT_p[j] \leqslant k < FT_p[j+1]$$

（为清楚起见，这里忽略了模运算。）

为了解释这种查找，我们从结点 1 解析 k=26，如图 5.4 所示。首先，结点 1 在其指状表中查找 k=26，发现该值大于 $FT_1[5]$，表示该请求将转发给结点 18（$=FT_1[5]$），而结点 18 将选择结点 20，因为 $FT_{18}[2] < k \leqslant FT_{18}[3]$。最后，该请求从结点 20 转发给结点 21，从结点 21 又转发给结点 28，该结点负责 k=26。此时，结点 28 的地址返回给结点 1，于是就完成了该键值的解析。同样，当请求结点 28 解析键值 k=12 时，该请求将按图 5.4 所示的虚线路由。可以证明，一个查找通常需要 O(log(N)) 步，其中 N 为系统中的结点数。

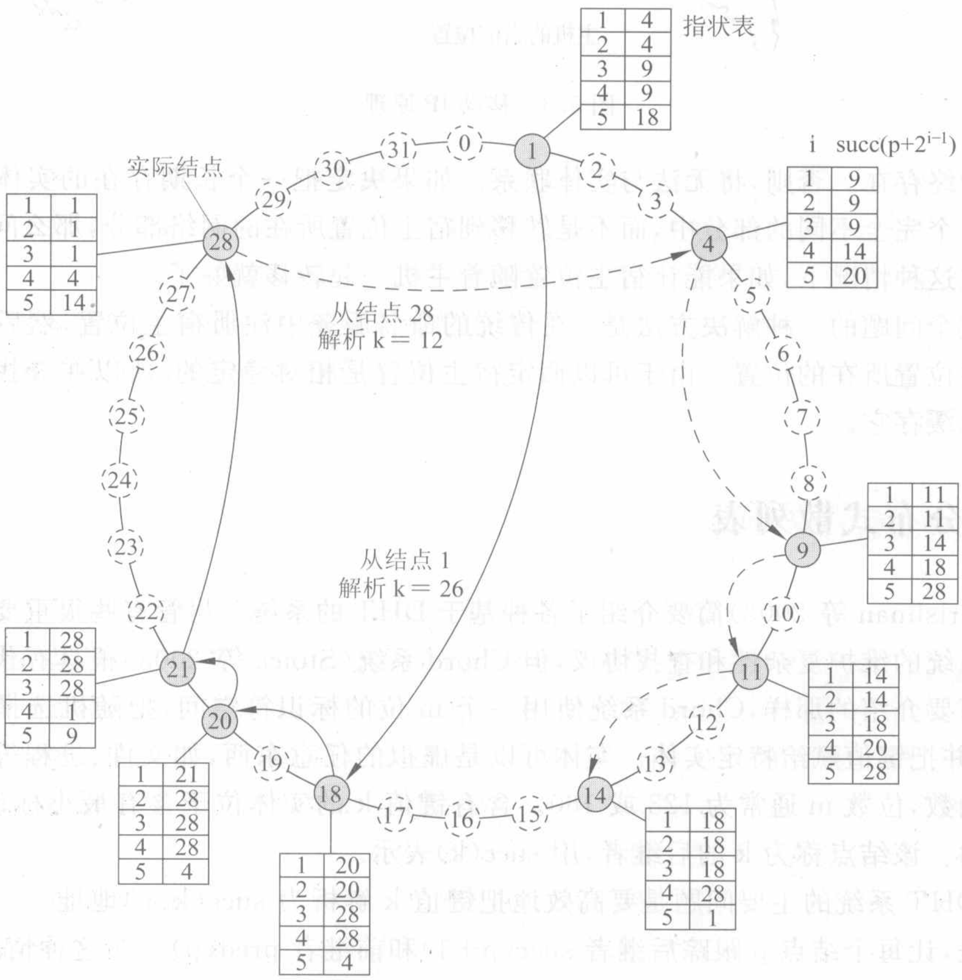


图 5.4 在 Chord 系统中从结点 1 解析键值 26，从结点 28 解析键值 12

在大型的分布式系统中，参与结点集可能总在变化。我们不仅要考虑到结点会随时加入和离开，而且还要考虑结点失效（结点离开了系统）以及随后又被恢复（重新加入系统）的情况。

加入诸如 Chord 的基于 DHT 的系统相对简单。假设结点 p 要加入。它只需与已有系统中的任意结点联系，并请求查找 succ(p+1)。一旦标识了该结点，p 就可以把自己插入到该环中。离开也同样简单。注意，结点还要跟踪其前继者。

显然，这里的复杂性来自保持指状表的更新。最重要的是，对每个结点 q，当实体指向环中的下一个结点时，$FT_q[1]$是正确的，即是后继者 q+1。为了达到这个目的，每个结点 q 定期地运行一个进程来与 succ(q+1)联系，并请求返回 pred(succ(q+1))。如果 q=pred(succ(q+1))，那么 q 就知道它的信息与其后继者的是一致的。否则，如果 q 的后继者更新了它的前继者，那么显然是有一个新结点 p 加入了该系统，且有 $q<p\leqslant succ(q+1)$，这样 q 就将 $FT_q[1]$调整为指向 p。此时，它还将检查 p 是否已把 q 记作为其前继者。如果没有，就需要再次调整 $FT_q[1]$。

同样，要更新指状表，结点 q 只需为每个实体 i 的 $k=q+2^{i-1}$找到后继者。这也可以通过发出一个解析 succ(k)的请求来完成。在 Chord 系统中，这种请求是通过一个后台进程来定期发出的。

同时，每个结点 q 将定期地检查其前继者是否为活动状态。如果前继者失效，q 所能做的唯一事情就是把 pred(q)记录为“未知”。另一方面，当结点 q 更新其链接指向环中的下一个已知结点，并发现 succ(q+1)的前继者已设置为“未知”时，它将通知 succ(q+1)，它是 succ(q+1)的前继者了。总之，这些过程确保了 Chord 系统总是一致的，只是出现了一些新结点。详细内容可参见(Stoica 等 2003)。

利用网络邻近

诸如 Chord 的系统的一个潜在问题是，请求跨因特网进行路由时不稳定。例如，假设图 5.4 的结点 1 位于荷兰的阿姆斯特丹，结点 18 位于美国加州的圣迭哥，结点 20 又位于阿姆斯特丹，结点 21 位于圣迭哥。解析键值 26 的结果是产生了三次广域消息传输，这可以减少为最多一次。为了最小化这些病态的情况，设计基于 DHT 系统时要求考虑到底层网络的因素。

(Castro 等 2002b)把考虑了底层网络的设计方法分为不同的三种。在**基于拓扑的结点标识符赋值**(topology-based assignment of node identifier)中，其思想是在标识符赋值时，两个邻近结点所赋给的标识符也是靠近的。不难想象，这种方法在诸如 Chord 之类的系统中会导致严重问题。在这种情况下，结点标识符是取自一维空间的，要把逻辑环映射到因特网很繁琐。而且，这种映射很容易产生相关联的失效：同一企业网络中的结点的标识符来自相对小的区间。当该网络不可达到时，均衡分布的标识符突然出现了一个间隔。

对**邻近路由**(proximity routing)，结点维护一个转发请求的可选列表。例如，在 Chord 习题中，每个结点不只有一个后继者，而是有 r 个后继者。事实上，这种冗余可以为指状表的每个实体应用。对结点 p，$FT_q[i]$指向$[p+2^{i-1}, p+2^i-1]$区间内的第一个结点。没有原因说明 p 为什么不能跟踪该区间内的 r 个结点：如果需要，其中的每一个都可以用来路由对键值 $k>p+2^i-1$ 的查询请求。在这种情况下，当结点选择转发一个查找请求时，可以选用离它最近的 r 个后继者中的一个，同时还满足如下限制：被选择结点的标识符应比所请求的键值的小。每个表实体有多个后继者的另一个优点是，结点的失效不会立即导致查询的失效，因为有多个路由可以使用。

最后，在**邻近邻结点选择**(proximity neighbor selection)中，其思想是优化路由表，使得选择离最近的结点作为邻结点。这种选择只在有更多的结点可供选择时才行。在 Chord 系统中，往往不是这种情况。但是，在诸如 Pastry(Rowstron 和 Druschel 2001)的其他协议

中，当结点加入系统时，会从其他多个结点接收有关当前覆盖网络的信息。新结点可用这些信息来构建一个路由表。显然，当还要其他结点可供选择时，邻近邻结点选择允许新加入的结点选择最好的一个。

注意，可能不太容易区分邻近路由和邻近邻结点选择。事实上，当把 Chord 系统修改为每个指状表实体可以包含 r 个后继者时，邻近邻结点选择采用的是标识最近的 r 个邻结点，这与刚刚解释的邻近路由很类似(Dabek 等 2004b)。

最后还要注意迭代和递归查询的区别。在迭代查询中，请求查找某个键值的结点将返回请求进程找到的下一个结点的网络地址。该进程然后请求下一结点采用另一步来解析键值。另一种方法是让结点把查询请求转发给下一个结点。这两种方法都有其优缺点，本章后面将进行介绍。

5.2.4 分层方法

本节我们首先讨论一种用于分层定位方案的一般方法，然后再提出多种优化方法。我们提供的方法基于 Globe 定位服务(Ballintijn 2003)。(van Steen 等 1998b)对这种服务进行了介绍。这是一种通用定位服务，是典型的分层定位服务，这些服务都是为个人通信系统(personal communication systems)提出的(Pitoura 和 Samaras 2001)。

在分层方案中，网络被划分为一组域(domain)。有一个覆盖整个网络的顶级域。每个域又可以进一步划分成多个更小的子域。最底层的域称为**叶域**(leaf domain)，叶域通常与计算机网络中的局域网相对应，或者与移动电话网络中的单元对应。

每个域 D 都拥有关联的目录结点 dir(D)。dir(D)会跟踪域中的实体，这样就形成了一棵目录结点树。顶级域的目录结点称为**根(目录)结点**(root (directory) node)，它包括了全部实体。图 5.5 显示了这种把网络划分为域和目录结点的总体组织方法。

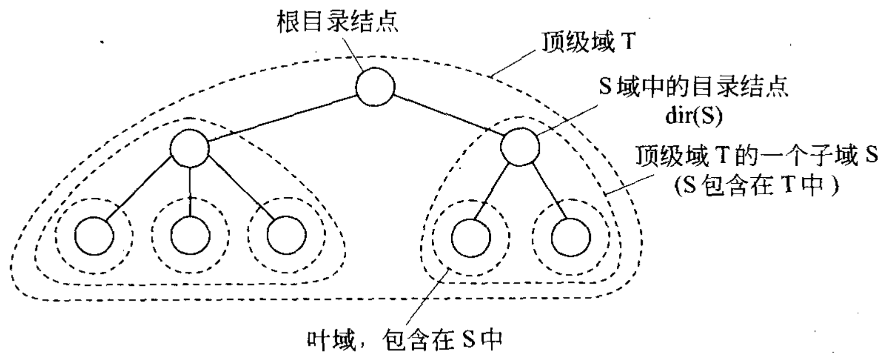


图 5.5 把定位服务划分为域的分层组织，每个域都拥有关联的目录结点

为了跟踪实体的位置，每个当前位于域 D 内的实体都由一条在目录结点 dir(D)中的**位置记录**(location record)来表示。在叶域 D 的目录结点 N 中，实体 E 的位置记录包含该实体当前在域中的位置。而含有 D 的、更高一级的叶域 D′使用目录结点 N′，N′也拥有一条关于 E 的位置记录，但是这一条记录只包含一个指向 N 的指针。同样，N′的父结点也将存储一条关于 E 的位置记录，这条位置记录也只包含一个指向 N′的指针。因此，根结点将拥有每个实体的位置记录，其中每条位置记录都存储一个指向更低层子域目录结点的指针，该记

录关联的实体当前就位于这里。

实体可以拥有多个地址，比如说它被复制了，就会出现这种情况。如果实体分别在叶域 D1 和 D2 中拥有地址，那么同时包含 D1 和 D2 的最小域的目录结点将有两个指针，每个指针都指向一个包含地址的子域。这样就产生了图 5.6 所示的通用组织树。

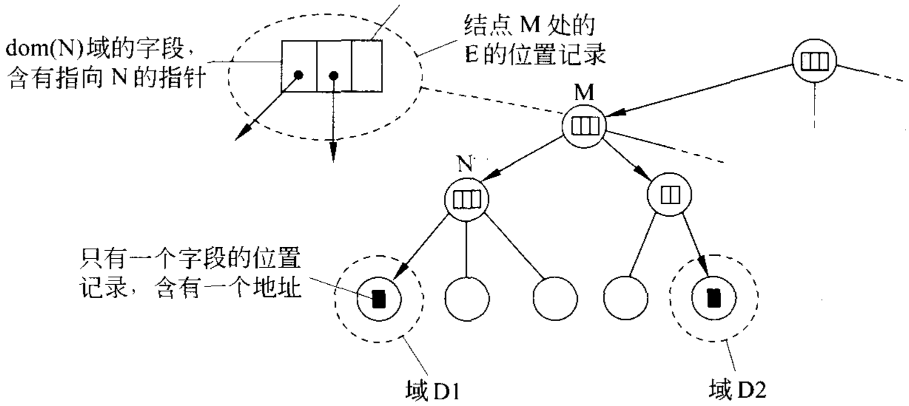


图 5.6　实体信息存储示例，这里的实体在不同的叶域中拥有两个地址

现在让我们考虑一下在这种分层定位服务中查询操作的实现方法。如图 5.7 所示，希望定位实体 E 的客户向它所在的叶域 D 的目录结点发送一个查找请求。如果这个目录结点没有存储该实体的位置记录，那么就说明该实体现在不在 D 中。因此，这个结点会把请求转发给它的父结点。注意，父结点代表一个比它的子域更大的域。如果父结点也没有 E 的位置记录，那么就会把该查找请求转发给更高一层的域，以此类推。

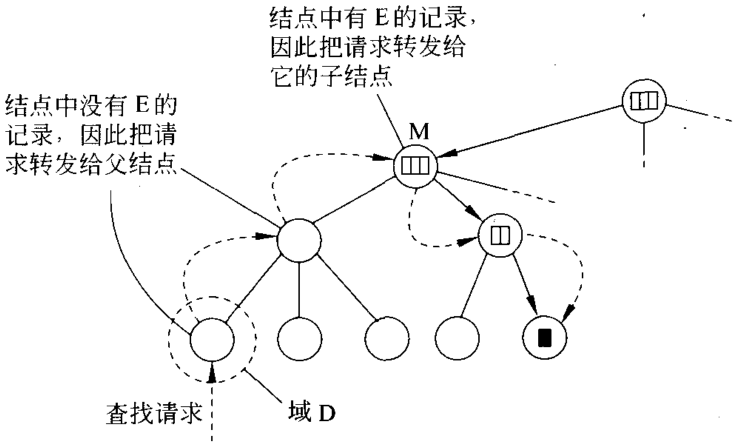


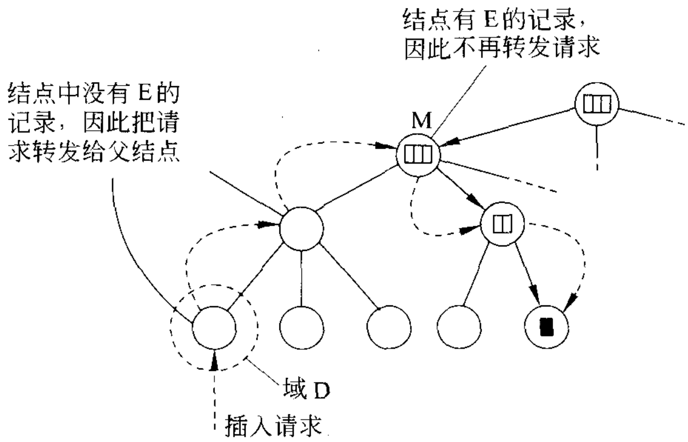
图 5.7　在分层组织的定位服务中的位置查找

如果结点 M 存储了 E 的位置记录，那么一旦请求到达 M 后，就可以知道 E 位于结点 M 代表的域 dom (M)中。如图 5.7 所示，M 存储了一条位置记录，其中包含一个指向其子域的指针。然后，把请求转发给那个子域的目录结点，那个子域会依次进一步向树的下方转发请求，直到请求最终到达叶结点为止。存储在叶结点中的位置记录会包含 E 在哪一个叶域中的地址。这样就可以把这个地址返回发起请求的客户。

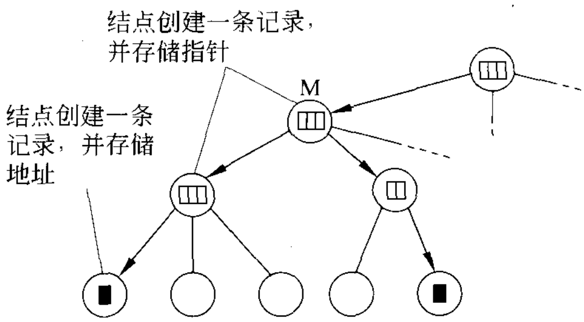
一个与分层定位服务有关的重要因素是，查找操作是在局部进行的。原则上，对实体的搜索是在一个以发出查找请求的客户为中心、逐步增大的环中进行的。每当查找请求被转

发到更高一层的目录结点时，都会扩大搜索区域。在最差的情况下，搜索会连续进行，直至请求到达根结点为止。由于根结点拥有所有实体的位置记录，所以此时可以简单地沿着一条向下的指针路线把请求转发给一个叶结点。

更新操作以类似的方式在局部进行，如图5.8所示。假设实体E在叶域D中创建了一个副本，就需要插入这个副本的地址。插入操作从D的叶结点dir(D)开始，然后D会立即把插入请求转发给它的父结点。父结点同样会转发插入请求，直到插入请求到达已经为E存储了位置记录的目录结点M为止。



(a)



(b)

图 5.8　更新操作

(a) 插入请求被转发到第一个知道实体E的结点；(b) 转发指向叶结点的指针所形成的链

然后，结点M会在E的位置记录中存储一个指针，该指针指向转发插入请求的那个子结点。此时，该子结点会建立一条关于E的位置记录，该位置记录中包含一个指针，指向转发请求的下一层结点。这个过程会连续进行，直至到达发起请求的叶结点为止。最后，那个叶结点会创建一条记录，这条记录包含实体在相关叶域中的位置。

刚才介绍的插入地址操作会导致一条从上到下建立的指针链，它从一个最底层的目录结点开始，该结点拥有实体E的位置记录。另一种是在向父结点发送插入请求之前创建一条位置记录。换句话说，就是从下往上建立指针链。后者的优点是地址会尽快变得可用。这样的话，即使父结点暂时无法到达，仍然可以在当前结点所在的域内查找实体的地址。

删除操作与插入操作类似。如果需要删除叶域D中实体E的地址，那么需要请求目录结点dir(D)从它为E存储的位置记录中删除该地址。如果那条位置记录变成了空的，也就是说，它不再包含其他E在D中的地址，那么这条记录也可以删除。在这种情况下，dir(D)

的父结点需要删除指向 dir(D)的指针。如果父结点中有关 E 的位置记录也变空了，那么也应该删除这条位置记录，并通知更高一层的结点。以此类推，这个过程将连续进行，直到从一条位置记录中删除指针后该位置记录仍然非空或者是到达根结点为止。

5.3 结构化命名

无层次名称适合于机器，但人类使用往往不方便。命名系统往往还支持结构化名称，该名称是由简单的、人类可阅读的名称组成。文件命名和因特网中主机的命名都使用这种方法。本节将介绍结构化名称，以及将这些名称解析为地址的方法。

5.3.1 名称空间

名称通常组织成**名称空间**(name space)。结构化名称的名称空间可以表示为带有标记的具有两种类型的结点的有向图。**叶结点**(leaf node)表示一个具名实体，它没有分支边。叶结点通常存储关于它所表示的实体的信息(例如实体的地址)，以便客户访问该实体。另外，叶结点还可以存储实体的状态，例如在文件系统中，叶结点包含了它所代表的完整文件。我们在下文中还会继续对结点内容的讨论。

与叶结点相比，**目录结点**(directory node)具有一定数量的分支边，每条边用一个名称来标记，如图 5.9 所示。命名图中的每个结点都认为是分布式系统中的一个实体，而且具有一个相关的标识符。目录结点用于存储一个表，其中每条分支边用一个(边标签，结点标识符)对来表示。这样的表称为**目录表**(directory table)。

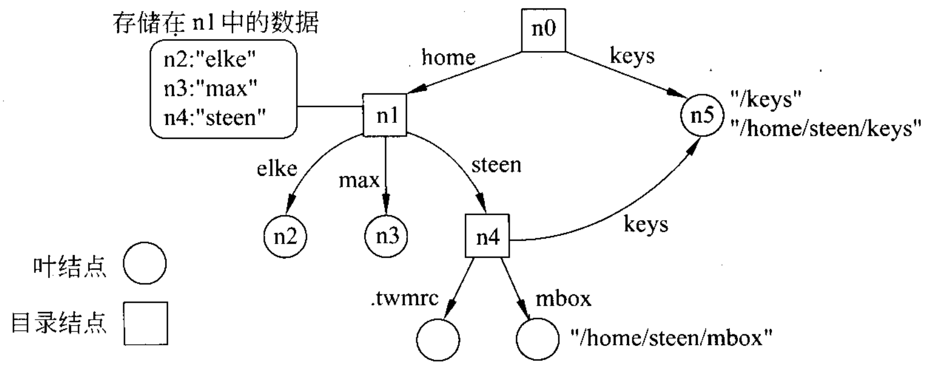


图 5.9 具有单个根结点的常见命名图

图 5.9 所示的命名图具有一个特殊的结点，即 n0，它只具有分支边，而没有进入边。这样的结点称为命名图中的**根**(**结点**)(root (node))。尽管命名图可以含有多个根结点，但为了简单起见，许多命名系统只含有一个根结点。命名图中的每个路径可以通过路径中的边的标签序列来指向，例如：

N:〈label-1，label-2，…，label-n〉

其中 N 指路径中的第一个结点。这样的序列称为**路径名**(path name)。如果路径名中的第一个结点是命名图的根，那么该路径名称为**绝对路径名**(absolute path name)，否则，称

为**相对路径名**(relative path name)。

认识到名称始终组织成名称空间这一点是很重要的。于是，名称始终仅相对于目录结点进行定义。在这个意义上，绝对名称这个术语很容易使人误解。同样，全局和局部名称之间的差异有时也会混淆。**全局名称**(global name)无论在系统中何处使用，都表示同一个实体。换句话说，全局名称始终根据相同的目录结点进行解释。而**局部名称**(local name)需要根据使用名称的位置来进行解释。当放置在不同的位置上时，局部名称实质上是相对名称，包含它的目录是隐式已知的。我们在讨论名称解析时还会回来讨论这些问题。

命名图的描述与其在许多文件系统中的实现是非常接近的。但是，在具体实现中，不是用边标记序列来表示路径名的。文件系统中的路径名通常用一个字符串来表示，其中边标记之间由特定的分隔字符来分隔，例如斜杠("/")。该字符也用来表明路径名是否是绝对的。例如，在图5.9中，通常不使用路径名n0：⟨home，steen，mbox⟩，而是使用其字符串表示法/home/steen/mbox来表示。还要注意的是，当有多个路径通向同一结点时，该结点可以由不同路径名来表示。例如，图5.9中的结点n5，使用/home/steen/keys和/keys都可以指向该结点。路径名的字符串表示法除了用于文件系统之外，也可以用于其他命名图中。在Plan 9(Pike等1995)中，所有的资源(例如进程、主机、I/O设备和网卡)都是以与传统文件相同的方式进行命名的。这个方法类似于为在一个分布式系统中的所有资源实现一个单一的命名图。

组织名称空间有许多不同的方法。如我们所提到的，大多数名称空间仅有一个根结点。在许多情况下，名称空间也严格地进行分层，将命名图组织成一棵树。这意味着除了根结点之外，每个结点都只有一条进入边，而根结点没有进入边。因此，每个结点只有一个相关联的绝对路径名。

图5.9所示的命名图是一个**有向无环图**(directed acyclic graph)的示例。在这样的组织结构中，一个结点可以具有多条进入边，但是图中不允许有循环。也有些名称空间没有此限制。

为了更具体一些，我们来考虑一下在传统UNIX文件系统中对文件进行命名的方式。在UNIX的命名图中，一个目录结点表示一个文件目录，而一个叶结点表示一个文件。唯一的根目录在命名图中用根结点表示。命名图的实现是文件系统的完整实现的一部分。该实现由逻辑磁盘上一系列连续的块组成，通常分为一个根块、一个超级块、一系列索引结点(称为信息结点，inode)和一些文件数据块。请参见(Crowley 1997；Silberschatz等2005；Tanenbaum和Woodhull 2006)。该组织结构如图5.10所示。

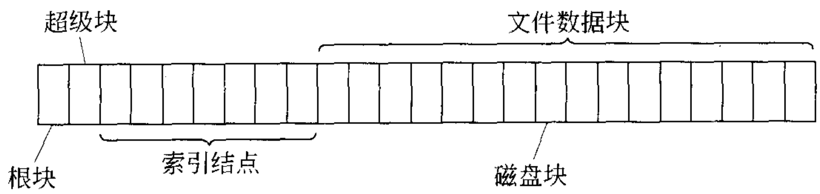


图5.10 在逻辑磁盘的连续磁盘块上实现UNIX文件系统的一般组织结构

根块是一个特殊的数据和指令块，在系统启动时自动装载到主存储器中。根块用于将操作系统装载到主存储器中。

超级块保存整个文件系统的信息，如文件系统的大小，磁盘上还有哪些块没有分配，以

及哪些信息结点还没有使用,等等。信息结点由索引号来引用,从数字0开始(数字0是为表示根目录的信息结点保留的)。

每个信息结点包含关于在磁盘上哪个位置可以找到与结点相关联的文件数据的准确信息。另外,一个信息结点包含关于其所有者、创建时间和最后一次修改的时间、保护信息等诸如此类的信息。因此,如果给定了一个信息结点的索引号,那么就可以访问到与其关联的文件。每个目录都作为一个文件来实现,对于根目录也是如此。它包含文件名与信息结点的索引号之间的映射。因此,信息结点的索引号可以看作相当于命名图中的结点标识符。

5.3.2 名称解析

名称空间为通过名称来存储和检索实体的信息提供了一套便利的机制。更一般地讲,给定一个路径名,应该能够查找出存储在由该名称所指向的结点中的任何信息。查询名称的过程称为**名称解析**(name resolution)。

为了解释名称解析是如何进行的,请考虑一个路径名,如N:〈label-1,label-2,…,label-n〉。该名称的解析是从命名图的结点N开始的,在其目录表中查询名称label-1,并返回label-1所指向的结点的标识符。然后,解析过程在标识出的结点的目录表中继续查询名称label-2,依此类推。假设该命名路径确实存在,那么解析会在由label-n指向的最后一个结点上停止,并返回该结点的内容。

名称查询从名称解析程序继续运行的地方返回结点的标识符。应特别指出的是,访问标识结点的目录表是必要的。再次考虑UNIX文件系统中的命名图。如前面所提到的,结点标识符是作为信息结点的索引号来实现的。访问目录表意味着首先要读取信息结点,以找到实际数据在磁盘上存储的位置,然后再读取含有目录表的数据块。

5.3.2.1 终止机制

仅当我们知道如何启动以及在何处启动时才能进行名称解析。在我们的实例中,开始结点已经给出,而且我们假设已经访问了目录表。知道如何启动以及在何处启动名称解析通常称为**终止机制**(closure mechanism)。从本质上讲,终止机制处理从名称空间中选择初始结点,名称解析从初始结点开始(Radia 1989)。有时难以理解终止机制的原因是它们必须是部分隐式的,而且进行比较时彼此截然不同。

例如,在UNIX文件系统命名图的名称解析中,根目录的信息结点是代表文件系统的逻辑磁盘的第一个信息结点。其实际的字节偏移量由超级块的其他字段中的值和操作系统中有关超级块的内部组织的硬性编码信息一起计算得出。

为了理解这个内容,请考虑文件名的字符串表示,如/home/steen/mbox。要解析该名称,需要事先访问适当命名图的根结点的目录表。作为根结点,结点自身不能被查询,除非它作为其他命名图中的不同结点来实现,譬如说G。但是在这种情况中,需要事先访问G的根结点。因此,解析文件名需要事先实现一些机制,通过它们解析程序才可以启动。

一个完全不同的示例是使用字符串"0031204430784"。许多人都不知道如何处理这些数字,除非他们被告知这个数列是一个电话号码。该信息足够启动解析程序,特别是通过拨打号码。然后,电话系统来完成剩下的工作。

最后我们考虑一个示例，即在分布式系统中使用全局和局部名称。局部名称的一个典型实例是环境变量。例如，在 UNIX 系统中，名为 HOME 的变量用于查询用户的主目录。每个用户都有该变量的副本，它初始化为与用户的主目录的全局系统名称相对应。与环境变量相关联的终止机制通过在特定于用户的表中查询变量的名称以确保该名称可被适当地解析。

5.3.2.2 链接和挂载

与名称解析紧密联系的是**别名**（aliases）的使用。别名是同一实体的另一个名称。环境变量是别名的一个例子。在命名图中，主要有两种不同的方法来实现别名。第一个方法是允许多个绝对路径名来指向命名图中的同一结点。该方法如图 5.9 所示，其中结点 n5 可以由两个不同的路径名来查询。在 UNIX 术语中，图 5.9 中的两个路径名/keys 和/home/steen/keys 都称为到结点 n5 的**硬链接**（hard link）。

第二种方法是用叶结点来表示实体（譬如说 N），而不是存储实体的地址或位置，该结点存储绝对路径名。当第一次解析指向 N 的绝对路径名时，名称解析将返回存储在 N 中的路径名，在这点它可以继续解析新的路径名。这个原理对应于 UNIX 文件系统中的符号链接（symbolic link），如图 5.11 所示。在这种情况中，指向包含绝对路径名/keys 的结点的路径名/home/steen/keys 是对结点 n5 的符号链接。

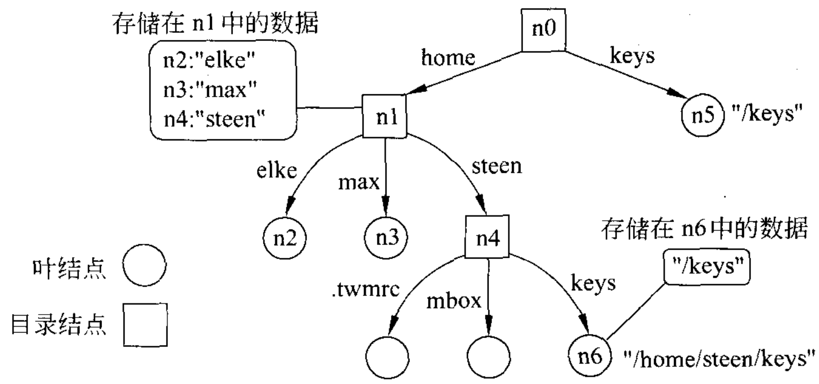


图 5.11 命名图中所讲述的符号链接的概念

到目前为止，所介绍的名称解析都是在单个名称空间中进行的。但是，名称解析也可以用来以透明方式来合并不同的名称空间。首先，让我们考虑一个已挂载的文件系统。在我们的命名模型中，已挂载的文件系统于让目录结点存储来自不同名称空间（我们称之为外部名称空间）的目录结点的标识符。存储结点标识符的目录结点称为**挂接点**（mount point）。因此，外部名称空间中的目录结点称为**挂载点**（mounting point）。通常，挂载点是名称空间的根。在名称解析的过程中，挂载点被查询，并通过访问目录表来完成解析。

挂载的原理可以推广到其他名称空间。作为挂接点的目录结点存储识别和访问外部名称空间中的挂载点的所有必需的信息。在许多分布式文件系统中实际上也是遵循该方法的。

请考虑分布在多台计算机上的名称空间的集合。特别是，每个名称空间由不同的服务器来实现，每个服务器可能运行在单独的计算机上。因此，如果想要将外部名称空间 NS2 挂载到名称空间 NS1 中，可能需要在网络上与 NS2 的服务器进行通信，因为服务器可能运

行在不同的计算机上而不是 NS1 的服务器上。在分布式系统中挂载外部名称空间至少需要以下信息：

(1) 访问协议的名称。

(2) 服务器的名称。

(3) 外部名称空间中挂载点的名称。

注意，这些结点每一个都需要解析。访问协议的名称需要解析成协议的执行，通过协议的执行可以与外部名称空间的服务器进行通信。服务器的名称需要解析成可以到达服务器的地址。作为名称解析中的最后部分，挂载点的名称需要解析成外部名称空间的结点标识符。

在非分布式系统中，不需要这三点中的任意一点。例如，在 UNIX 中，没有访问协议和服务器。同样，挂载点的名称也是不需要的，它仅是外部名称空间的根目录。

挂载点的名称由外部名称空间的服务器来解析。但是，我们还需要名称空间，以及访问协议的实现和服务器名称。一个可能性是将上面所列的三个名称表示为一个 URL。

为了使概念更具体，请考虑这样一种情况，使用笔记本电脑的用户想要访问远程文件服务器上存储的文件。客户计算机和文件服务器都使用 Sun 公司的**网络文件系统**(Network File System，NFS)来配置，我们将在第 11 章中对 NFS 进行讨论。NFS 是一种分布式文件系统，它使用一种明确描述客户如何访问存储在远程 NFS 文件服务器上的文件的协议。特别是，如果允许 NFS 通过因特网来工作，那么客户可以通过 NFS URL 来确切指定它所要访问的文件，例如 nfs：//flits. cs. vu. nl//home/steen。该 URL 命名 NFS 文件服务器 flits. cs. vu. nl 上名为/home/steen 的文件(对目录也是一样)，客户可以通过 NFS 协议来访问它(Shepler 等 2000)。

就如何解释 nfs 名称这个问题，在全世界范围内已经达成协议，从这个意义上来说，nfs 是一个众所周知的名称。换句话说，如果我们正在处理 URL，那么名称 nfs 将解析成 NFS 协议的实现。使用 DNS 可以把服务器名称解析成它的地址，我们将在后面章节中讨论域名系统。正如我们说过的那样，/home/steen 由外部名称空间的服务器进行解析。

图 5.12 部分显示了在客户机器上的文件系统的组织方式。根目录包含许多用户定义的条目，其中有名为/remote 的子目录。这个子目录用来包含外部名称空间的挂载点，例如 Vrije Universiteit 上的用户根目录。为此，可使用一个名为/remote/vu 的目录结点来存储 URL nfs：//flits. cs. vu. nl/home/steen。

现在来考虑名称/remote/vu/mbox。在解析这个名称时，将从客户机器的根目录开始，一直到结点/remote/vu 为止。然后名称解析会返回 URL nfs：//flits. cs. vu. nl//home/steen，接着会引导客户机器使用 NFS 协议与文件服务器 flits. cs. vu. nl 联系，随后会访问目录/home/steen。接下来的名称解析将读取该目录下的 mbox 文件。

举例来说，刚才描述的允许挂载远程文件系统的分布式系统允许客户机器执行如下命令：

```
cd /remote/vu
ls-1
```

随后这些命令将列出远程文件服务器上/home/steen 目录下的文件。这样做的优点是

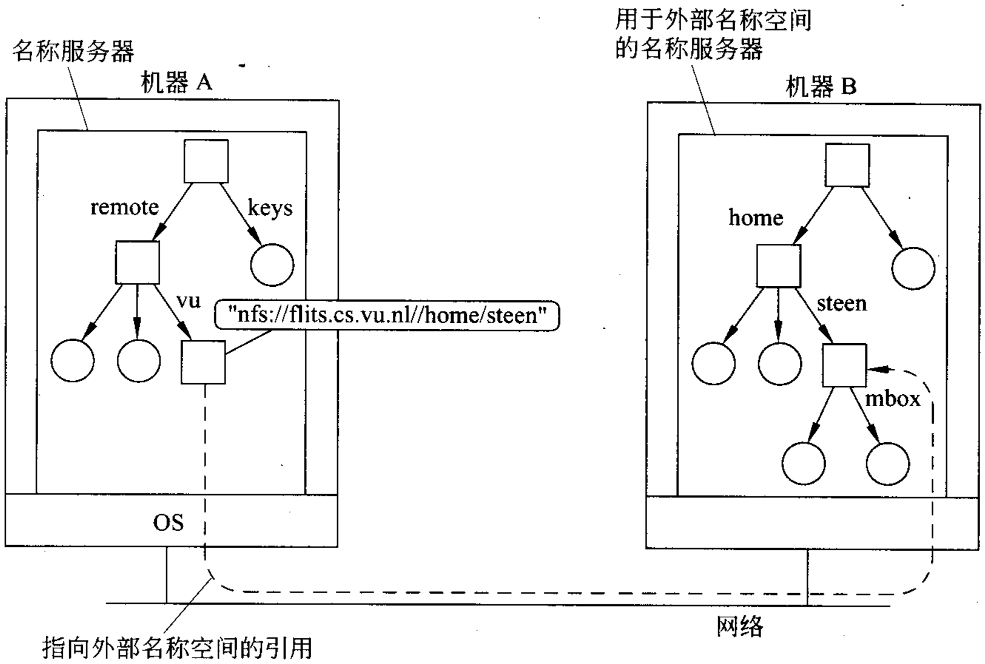


图 5.12 通过指定的访问协议装配远程名称空间

用户不必了解访问远程服务器的实际细节。在理想情况下，与访问本地可用文件相比，仅仅存在一些性能损失。实际上，在客户看来，来自于本地机器的名称空间和来自远程机器上/home/steen 下的名称空间共同形成了一个单一的名称空间。

5.3.3 名称空间的实现

名称空间形成了命名服务的核心，准确地说，命名服务就是一种允许用户和进程添加、删除和查找名称的服务。命名服务由名称服务器实现。如果一个分布式系统局限于局域网，那么可以只使用一台名称服务器来实现命名服务。可是，大型分布式系统包含许多实体，可能会跨越很大的地理区域，在这样的系统中，需要使用多台名称服务器来分散名称空间的实现。

5.3.3.1 名称空间的分布性

在大型的、可能是世界范围的分布式系统中，名称空间通常分层组织。和前面一样，仍然假定这样的名称空间只有一个根结点。为了有效地实现这样的名称空间，把它分成逻辑上的层会很方便。(Cheriton 和 Mann 1989)把名称空间区分为以下三层。

全局层(global layer)由最高级别的结点组成，即由根结点以及其他逻辑上靠近根结点的目录结点，也就是根结点的子结点组成。全局层的特点就是它通常是稳定的，也就是说目录表很少改变。这样的结点可以代表组织或组织群，这是因为它们的名称被存储在这个名称空间中。

行政层(administrational layer)由那些在单个组织内一起被管理的目录结点组成。行政层中的目录结点所具有的特点是它们代表属于同一组织或行政单位的实体组。举例来说，这样的目录结点可能是一个组织中每个部门都拥有的目录结点，或者是通过它可以找到

所有主机的目录结点。还有可以用来命名所有用户起点的目录结点，等等。行政层的结点是相对稳定的，尽管说它们的改变一般要比全局层结点频繁得多。

最后是**管理层**(managerial layer)，它由那些经常改变的结点组成。例如，代表本地网络主机的结点就属于这一层。同样，本层还包括那些代表共享文件(如库文件或二进制文件)的结点。另一类重要结点包括那些代表用户定义目录和文件的结点。与全局层和行政层相比，管理层的结点不仅要由系统管理员维护，而且还要由分布式系统的各个最终用户维护。

为了更具体地介绍这些概念，图 5.13 显示了一个例子，它说明了把 DNS 名称空间划分为若干部分的方法，该空间包括了组织内文件的名称，可以通过因特网访问这些文件，如 Web 页面和可传输文件等。名称空间划分成不重叠的几部分，在 DNS 中称为区域(Zone)(Mockapetris 1987)。区域是名称空间的一部分，它是由单独的名称服务器实现的。图 5.13 展示了一些区域。

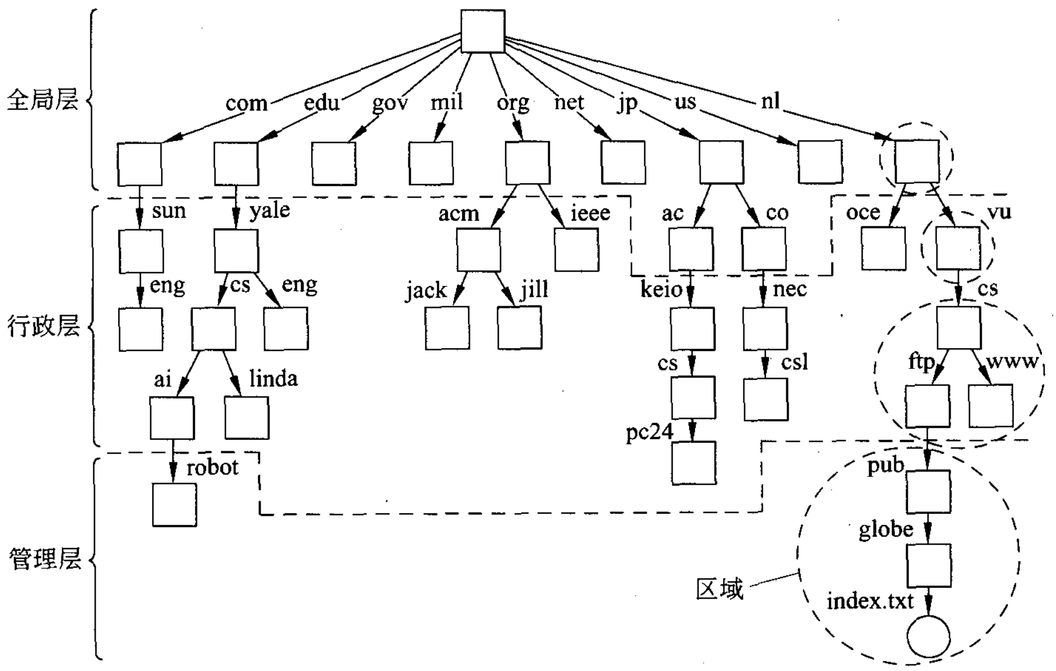


图 5.13　DNS 名称空间划分示例，该空间包括了可以从因特网上访问的文件，共分三层

在可用性和性能方面，每层的名称服务器都必须满足不同的要求。在全局层中，良好的可用性对名称服务器特别重要。如果名称服务器发生故障，那么名称空间中很大部分将无法到达，这是因为不能越过发生故障的服务器进行名称解析。

性能问题有点微妙。根据全局层结点变化不频繁的特点，查找操作的结果一般会长期有效。因此，客户可以有效地缓存(也就是本地存储)这些结果。当再次执行同样的操作时，可以从客户的缓存中直接获取结果，而不用让名称服务器返回结果。因此，全局层的名称服务器并不需要快速响应单一的查询请求。相反，重要的可能是其吞吐能力，特别是在拥有上百万用户的大型系统中。

将复制服务器与客户端缓存结合起来可以满足全局层名称服务器在可用性和性能方面的要求。正如我们将在第 7 章讨论的那样，这一层的更新通常不需要立即生效，让复制保持一致会使事情更为轻松一些。

对于与名称服务器同在一个组织内的客户来说，行政层名称服务器的可用性是最为重

要的。如果名称服务器发生故障，那么组织内的许多资源将因无法查找而变得不可到达。然而，对于组织外的用户来说，暂时无法到达组织内的资源可能没有那么重要。

在性能方面，行政层的名称服务器与全局层的名称服务器有类似的特点。由于结点的变化不是经常发生，所以缓存查询结果可能是非常有效的，这使得性能不那么重要。不过，与全局层相比，行政层应该注意查询结果在几 ms 内返回的情况，不管是直接从服务器返回，还是从客户的本地缓存返回。同样，更新处理一般应该比全局层更迅速。举例来说，需要几个小时来让新用户账号生效是不能接受的。

通过使用高性能的机器来运行名称服务器，上述要求一般可以得到满足。另外，应该采用客户端缓存的方法，并与复制相结合，从而提高整体可用性。

对管理层名称服务器的可用性要求一般比较低。特别是，使用一台专用的机器运行名称服务器来应付临时不可用的风险通常已经足够了。不过，在这里，性能是关键。用户希望操作能够立即完成。由于经常进行更新，所以客户端缓存一般不太有效，除非采用特殊措施，第 7 章将对此进行讨论。

图 5.14 显示了不同层的名称服务器之间的比较。在分布式系统中，全局层和行政层的名称服务器非常难以实现，困难来自于复制和缓存。复制和缓存对于提高可用性和性能来说是必需的，但它们也导致了一致性问题的出现。跨越广域网络的缓存和复制使问题进一步恶化，这是因为广域网络造成了较长的通信延迟，从而使同步变得更加困难。第 7 章将全面地讨论了复制和缓存。

内容	全局层	行政层	管理层
网络的地理范围	世界范围	组织	部门
结点数量	少许	多	极多
查询响应	s	ms	立即
更新的传播情况	延迟	立即	立即
复制数量	许多	没有或很少	没有
是否采用客户端缓存	是	是	有时

图 5.14 为大型名称空间所属的全局层、行政层和管理层实现结点的名称服务器之间的比较

5.3.3.2 名称解析的实现

在多个名称服务器上名称空间的分散性影响了名称解析的实现。为了说明名称解析在大型名称服务中的实现，我们假定不复制名称服务器，也不使用客户端缓存。每个客户都使用本地的名称解析程序，名称解析程序负责确保名称解析过程得以执行。请参考图 5.6，假定绝对的路径名

```
root:⟨nl,vu,cs,ftp,pub,globe,index.txt⟩
```

将被解析。如果使用 URL 表示法，那么这个路径名将对应于 ftp://ftp.cs.vu.nl/pub/globe/index.txt。现在可以用两种方法实现名称解析。

在**迭代名称解析**(iterative name resolution)中，名称解析程序把完整的名称转发给根

名称服务器。假定可以联系根服务器的地址是众所周知的。根服务器会尽量深入地解析路径名，然后把结果返回给客户。在我们的例子中，根服务器只能解析标识符 nl，然后根服务器将返回与 nl 相关的名称服务器所用的地址。

接着，客户会把剩下的路径名（也就是 nl：〈vu，cs，ftp，pub，globe，index. txt〉）发送给一台名称服务器。这台服务器只能解析标识符 vu，然后返回相关名称服务器的地址，同时剩下路径名 vu：〈cs，ftp，pub，globe，index. txt〉。

接下来客户的名称解析程序将同下一台名称服务器联系，该服务器响应的结果是解析标识符 cs，随后还有 ftp，最后会连同路径名 ftp：〈pub，globe，index. txt〉一起返回该 FTP 服务器的地址。客户端接着再与该 FTP 服务器联系，请求它解析原始路径名的最后一部分。随后 FTP 服务器会解析标识符 pub、globe 和 index. txt，并发送被请求的文件（这里是使用 FTP）。图 5. 15 显示了这个迭代名称解析过程。符号 #〈cs〉用来指明一台服务器的地址，该服务器负责处理〈cs〉涉及到的结点。

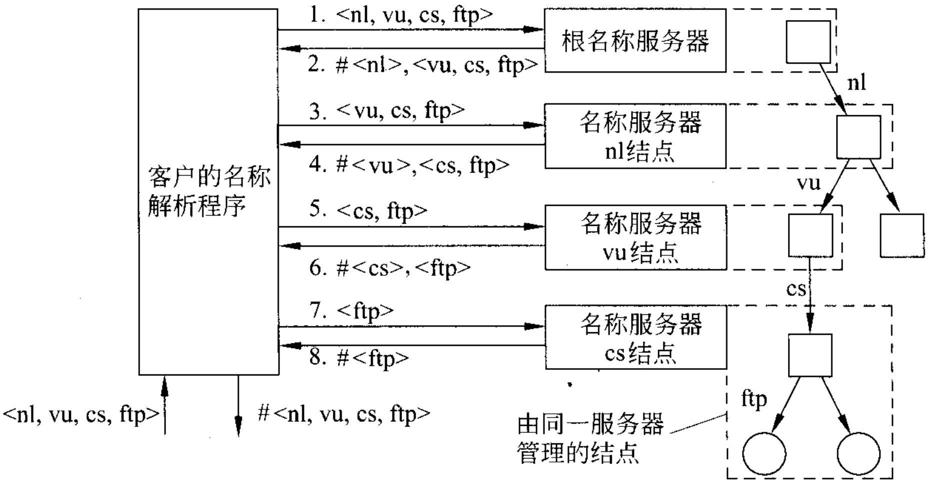


图 5. 15　迭代名称解析原理

实际上，最后一步，即联系 FTP 服务器并请求它发送路径名 ftp：〈pub，globe，index. txt〉所指文件的那一步，是由客户进程独立完成的。换句话说，客户通常只把路径名 root：〈nl，vu，cs，ftp〉转交给名称解析程序，希望从名称解析程序那里得到可以联系的那一台 FTP 服务器的地址，图 5. 15 也说明了这一点。

迭代名称解析的一种替代方法是在名称解析过程中使用递归。与返回客户解析程序每个中间结果不同，在使用**递归名称解析**（recursive name resolution）时，名称服务器会把结果传递给它找到的下一台服务器。举例来说，当根名称服务器找到实现 nl 结点的名称服务器所用的地址后，它会请求该名称服务器解析路径名 nl：〈vu，cs，ftp，pub，globe，index. txt〉。在同样使用递归名称解析的情况下，下一台服务器将解析完整的路径名，最终会把 index. txt 文件返回根服务器，根服务器再依次把该文件传递给客户的名称解析程序。

图 5. 16 说明了递归名称解析。和迭代名称解析一样，最后的解析步骤，即联系 FTP 服务器并请求它传送指定文件的那一步，一般情况下是由客户的进程独立实现。

递归名称解析的主要缺点是它要求每台名称服务器都具有较高的性能。本质上，需要名称服务器对路径名进行完整的解析，尽管说它可能会与其他名称服务器一起完成这项工

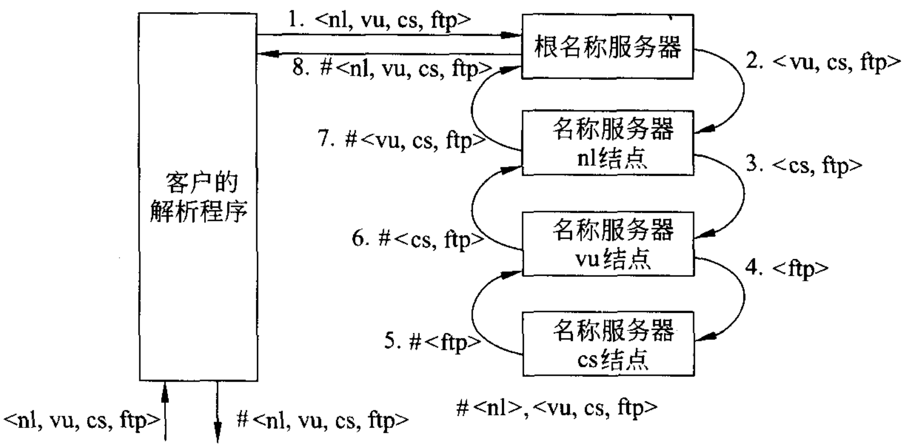


图 5.16　递归名称解析原理

作。这种额外的负担非常大，以至于在名称空间的全局层中，名称服务器只支持迭代名称解析。

递归名称解析拥有两个重要的优点。第一个优点就是与迭代名称解析相比，递归名称解析的缓存结果要更为有效一些。第二个优点是可以减少通信开销。为了说明这些优点，我们假定客户的名称解析程序接受的路径名只涉及名称空间中全局层或行政层结点。为了解析路径名中对应于管理层结点的那一部分，客户将独立联系名称解析程序返回的名称服务器，正如我们上面讨论的那样。

递归名称解析让名称服务器逐步负责实现更底层结点名称解析的名称服务器的地址。其结果是，可以有效地使用缓存来提高性能。例如，在请求根服务器解析路径名 root：〈nl，vu，cs，ftp〉时，根服务器最终会得到一台服务器的地址，这台服务器负责实现那个路径名所涉及的结点的名称解析。为了达到这一点，实现 nl 结点的名称服务器必须查找实现 vu 结点的名称服务器所用的地址，而后者必须查找实现 cs 结点的名称服务器所用的地址。

由于全局层和行政层结点不会经常改变，所以根名称服务器可以有效地缓存返回的地址。此外，由于地址也被递归返回负责实现 vu 结点以及 nl 结点的名称服务器，所以同样可以在这些服务器上缓存地址。

同样，也可以返回和缓存中间名称查询结果。例如，实现 nl 结点的服务器必须查找实现 vu 结点的服务器的地址。当实现 nl 的服务器返回原始名称的查询结果时，可以把地址返回根服务器。图 5.17 显示了解析过程的完整概括，以及各个名称服务器可以缓存的结果。

从根本上说，这种方法的优点就是可以非常有效地完成查询操作。例如，假设另一个客户后来请求解析路径名 root：〈nl，vu，cs，flits〉。该名称被传递给根服务器，根服务器可以立即把它转发给用于 cs 结点的名称服务器，并请求它解析剩下的路径名 cs：〈flits〉。

如果使用迭代名称解析，那么就需要把缓存限制在客户的名称解析程序上。因此，如果客户 A 请求解析一个名称，而另一个客户 B 随后又请求解析同一个名称，那么 B 请求的名称解析所经过的名称服务器将与 A 相同。作为一种折衷方案，许多组织都使用一台本地的中间名称服务器，所有客户都共享它。这台本地名称服务器处理所有的名称请求并缓存结

服务器所在的结点	需要解析的标识符	查询	传递给下一个服务器	递归与缓存	返回请求者
cs	<ftp>	#<ftp>	—	—	#<ftp>
vu	<cs,ftp>	#<cs>	<ftp>	#<ftp>	#<cs> #<cs， ftp>
nl	<vu,cs,ftp>	#<vu>	<cs,ftp>	#<cs> #<cs,ftp>	#<vu> #<vu,cs> #<vu,cs,ftp>
root	<nl,vu,cs,ftp>	#<nl>	<vu,cs,ftp>	#<vu> #<vu,cs> #<vu,cs,ftp>	#<nl> #<nl,vu> #<nl,vu,cs> #<nl,vu,cs,ftp>

图 5.17　对〈nl,vu,cs,ftp〉进行的递归名称解析。名称服务器缓存用于后续查询的中间结果

果。从管理的角度来看，使用这样一台中间服务器也是很方便的。例如，只有这台服务器才需要知道根服务器所在的位置，其他机器都不需要这种信息。

递归名称解析的第二个优点就是它在通信方面的开销通常较小。同样，设想要解析路径名 root:〈nl,vu,cs,fp〉，并且假设客户位于旧金山。假设客户知道用于 nl 结点的服务器的地址，在使用递归名称解析的情况下，通信路线从位于旧金山的客户主机开始到位于 The Netherlands 的 nl 服务器为止，如图 5.18 中的 R1 所示。从那里开始，依次需要 nl 服务器与 Vrije Universiteit 名称服务器（位于 The Netherlands 的 Amsterdam 一家大学的校园内）之间的通信。它们之间的通信如图 5.18 中的 R2 所示。最后，需要 vu 服务器与位于 Computer Science Department 的名称服务器之间的通信，如图 5.18 中的 R3 所示。返回路线是一样的，不过方向相反。很明显，决定通信开销的是客户主机与 nl 服务器之间的信息交换。

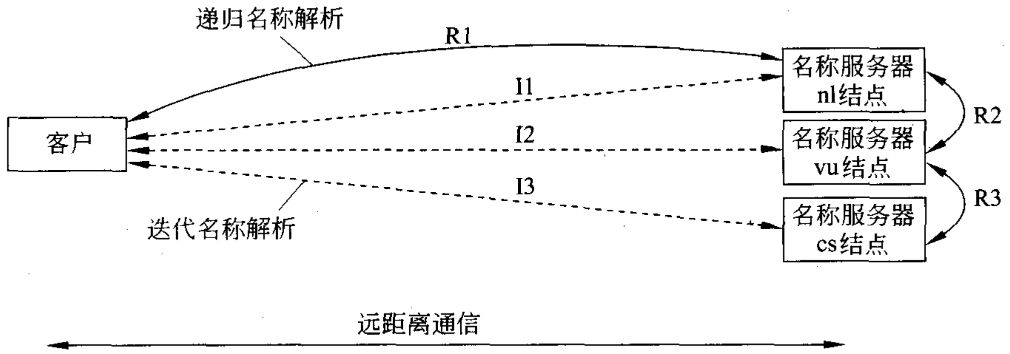


图 5.18　递归名称解析与迭代名称解析在通信开销方面的对比

与之相反，在使用迭代名称解析的情况下，客户主机必须分别与 nl、vu 和 cs 等服务器进行通信，这种通信的总开销大约是递归名称解析的通信开销的 3 倍。图 5.18 中标有 I1、I2 和 I3 的箭头显示了迭代名称解析的通信路线。

5.3.4　示例：域名系统

当今使用的最大分布式名称服务之一是因特网的**域名系统**（domain name system,

DNS)。DNS 主要用来查找主机地址和邮件服务器。在下面的内容中，我们将集中介绍DNS 名称空间的组织结构以及存储在其结点上的信息。另外，我们还要深入了解 DNS 的实际实现。有关这方面的更多信息，请参考(Mockapetris 1987；Albitz 和 Liu 2001)。对DNS 的最新评估，主要关注的是它是否仍能满足当前因特网的需要(Levien 2005)。从这个报告中可以得出一个令人吃惊的结论是，即使是 30 年以后，DNS 仍没有需要被替代的迹象。

5.3.4.1 DNS 名称空间

DNS 名称空间是分层组织的，就像一棵有根的树。标识符是区分大小写的字符串，该字符串由字母和数字组成。标识符的最大长度是 63 个字符，完整路径名的最大长度是 255 个字符。路径名的字符串表示由一列标识符组成，从最右边开始，使用圆点“.”分隔各个标识符。根由圆点代表。举例来说，路径名 root：〈nl，vu，cs，flits〉用字符串 flits. cs. vu. nl. 表示，这个字符串包括最右边的圆点，该圆点象征着根结点。为了方便阅读，我们通常省略这个圆点。

由于 DNS 名称空间的每个结点都包含一个进入边(根结点除外，它不包含进入边)，所以结点进入边的标识符也可用作该结点的名称。子树被称为**域**(domain)，指向根结点的路径名称为**域名**(domain name)。注意，域名可以是绝对的，也可以是相对的，就像路径名一样。

结点的内容由一组**资源记录**(resource record)组成。有各种不同类型的资源记录，图 5.19 列出了最重要的一些记录。

记录类型	相关实体	描述
SOA	区域	容纳表示区域的信息
A	主机	容纳结点所代表主机的 IP 地址
MX	域	指向邮件服务器，该服务器处理发往该结点的邮件
SRV	域	指向处理某种服务的服务器
NS	区域	指向实现指定区域的名称服务器
CNAME	结点	包含指定结点主名称的符号链接
PTR	主机	容纳主机的规范名称
HINFO	主机	容纳结点所代表主机的信息
TXT	任意类型	容纳任何被认为有用的、特定于实体的信息

图 5.19 形成 DNS 名称空间结点内容的一些最重要的资源记录类型

DNS 名称空间的结点经常同时代表几个实体。例如，像 vu. nl 这样的域名可以用来表示域和区域。在这种情况下，域由几个区域实现。

SOA(start of authority，即授权起点)资源记录包含的信息有：负责区域的系统管理员的电子邮件地址，可以从它上面获取区域数据的主机名称，等等。

A(address，即地址)记录代表因特网上的特定主机。A 记录包含允许通信的主机所用的 IP 地址。如果主机拥有多个 IP 地址，那么与多宿主计算机一样，该结点将包含每个地址的 A 记录。

MX(mail exchange，即邮件交换)记录是一种重要的资源记录类型，它本质上是指向代

表一台邮件服务器的结点的符号链接。例如，代表域 cs. vu. nl 的结点拥有一条包含名称 zephyr. cs. vu. nl 的 MX 记录，名称 zephyr. cs. vu. nl 指向了一台邮件服务器。该服务器会处理所有发给域 cs. vu. nl 用户的邮件。一个结点中可以存储多条 MX 记录。

SRV 记录与 MX 记录有关，SRV 记录包含处理特定服务的服务器名称。(Vixie 1996) 描述了 SRV 记录。服务本身由名称和协议名一起标识。例如，cs. vu. nl 域的 Web 服务器可以用一条类似于 http. tcp. cs. vu. nl 的 SRV 记录来命名。这条记录指向服务器的真正名称(即 soling. cs. vu. nl)。

代表区域的结点包含一条或多条 NS(name server，名称服务器)记录。和 MX 记录类似，NS 记录包含一个服务器的名称，该服务器实现该结点代表的区域。原则上，名称空间的每个结点都可以存储一条 NS 记录，这条 NS 记录指向实现结点的名称服务器。不过，正如我们下面将要讨论的那样，在 DNS 名称空间的实现中，只有代表区域的结点才需要存储 NS 记录。

DNS 区分别名和**规范名称**(canonical name)。每个主机都假定拥有一个规范名称，或者说主名。别名由存储 CNAME 记录的结点实现，CNAME 记录包含主机的规范名称。因而存储这种记录的结点名称与符号链接相同，如图 5.11 所示。

DNS 使用 PTR(pointer，指针)记录来维护 IP 地址到主机名称之间的反向映射。为了在只有 IP 地址的情况下提供主机名称查询，DNS 维护着一个名为 in-addr. arpa 的域，这个域中的结点代表因特网主机，这些结点使用代表主机的 IP 地址命名。例如，假设主机 www. cs. vu. nl 的 IP 地址是 130. 37. 25. 11。DNS 会创建一个名为 11. 25. 37. 130. in-addr. arpa 的结点，这个结点用来在 PTR 记录中存储主机的规范名称(正好是 soling. cs. vu. nl)。

最后两种记录类型是 HINFO 记录和 TXT 记录。HINFO(host info，主机信息)记录用来存储主机的其他信息，如机器类型和操作系统等。与此类似，TXT 记录用来存储任意类型的数据，存储这些结点所代表实体的有关信息对用户会有所帮助。

5.3.4.2 DNS 的实现

DNS 的实现与我们在前面小节中描述的那些实现方式很类似。本质上，DNS 名称空间可以划分成如图 5.13 所示的全局层和行政层。管理层一般由本地文件系统组成，在形式上它不是 DNS 的一部分，因此也不使用 DNS 来管理。

每个区域都由一个名称服务器来实现，从可用性考虑，这个名称服务器常常被复制。区域更新一般由主名称服务器完成。通过修改主服务器的 DNS 数据库，可以完成更新操作。辅名称服务器不直接访问该数据库，相反，它请求主服务器把内容传送给它。后一种做法在 DNS 术语中称为区域传送。

DNS 数据库由一组(数量很少)文件实现，其中最重要的一个文件包含了特定区域中所有结点的所有资源记录。这种方法允许只用结点的域名来标识结点，这样，结点标识符就变成了一个(隐含的)文件索引。

为了更好地理解这些实现方式，图 5.20 显示了一个文件的一部分，这个文件包含了有关 cs. vu. nl 域的大多数信息要注意，为了直观，我们对该文件进行了编辑。该文件显示了 8 个不同结点的内容，这些结点是 cs. vu. nl 域的一部分，在这个域中，每个结点都由它的域名来标识。

名称	记录类型	记录值
cs.vu.nl	SOA	star 1999121502,7200,3600,2419200,86400
cs.vu.nl	NS	star.cs.vu.nl
cs.vu.nl	NS	top.cs.vu.nl
cs.vu.nl	NS	solo.cs.vu.nl
cs.vu.nl	TXT	“Vrije Universiteit Math. & Comp. Sc.”
cs.vu.nl	MX	1 zephyr.cs.vu.nl
cs.vu.nl	MX	2 tornado.cs.vu.nl
cs.vu.nl	MX	3 star.cs.vu.nl
star.cs.vu.nl	A	
star.cs.vu.nl	A	
star.cs.vu.nl	MX	
star.cs.vu.nl	MX	
star.cs.vu.nl	HINFO	
zephyr.cs.vu.nl	A	
zephyr.cs.vu.nl	A	
zephyr.cs.vu.nl	MX	
zephyr.cs.vu.nl	MX	
	HINFO	
ftp.cs.vu.nl	CNAME	
www.cs.vu.nl	CNAME	
soling.cs.vu.nl	A	
soling.cs.vu.nl	A	
soling.cs.vu.nl	MX	
soling.cs.vu.nl	MX	
	HINFO	
vucs-das1.cs.vu.nl	PTR	
vucs-das1.cs.vu.nl	A	
inkt.cs.vu.nl	HINFO	
inkt.cs.vu.nl	A	
pen.cs.vu.nl	HINFO	
pen.cs.vu.nl	A	
localhost.cs.vu.nl	A	

图 5.20 区域 cs. vu. nl 使用的 DNS 数据（摘录）

结点 cs. vu. nl 既代表区域，也代表域。它的 SOA 资源记录包含了该文件的有效性信息，以后我们不必再关心这个文件。这个区域有三个名称服务器，可使用 NS 记录中的规范主机名引用它们。TXT 记录用来提供这个区域的其他信息，不过任何一台名称服务器都不能自动对它们进行处理。此外，还有三个邮件服务器，它们可以处理发给该域用户的邮件。邮件服务器名称前面的数字指定了选择顺序。发送邮件的服务器始终会首先与编号最小的邮件服务器联系，在这个例子中，也就是 zephyr. cs. vu. nl。

主机 star. cs. vu. nl 作为这个区域的一个名称服务器运行。名称服务器对于任何名称服务都是很关键的。可以看到，为了增强这个服务器的健壮性，提供了两个分开的网络接口，每一个都由一条独立的 A 资源记录来代表。使用这种方式可以避免网络连接中断带来的影响。

接下来的 4 行提供了有关邮件服务器的必要信息。要注意，这台邮件服务器也由其他邮件服务器提供备份，其路径是 tornado. cs. vu. nl。

下面6行显示了一种典型的配置，一台单独的机器使用这种配置实现了部门的 Web 服务器和 FTP 服务器，这台机器名为 soling. cs. vu. nl。通过在同一台机器上运行这两种服务器(本质上是只把该机器用于因特网服务)，系统管理可以变得更为简单一些。例如，两种服务器将包含相同的文件系统视图，从效率角度考虑，文件系统的一部分可以在 soling. cs. vu. nl 上实现。这种方法通常应用于 WWW 和 FTP 服务中。

下面两行显示的信息与部门的旧服务器集群有关。在这个例子中，它告诉我们地址 130.37.198.0 与主机名 vucs-das1. cs. vu. nl 相关联。

下面4行显示了连接到局域网的两台打印机的信息。注意，从 192.168.0.0 到 192.168.255.255 的地址段是私有的：这些地址只能从本地网络访问，因特网主机不能访问。

因为 cs. vu. nl 域被实现为单个区域，所以图 5.20 不包含对其他区域的引用。对在另一个区域里实现的子域中的结点，其引用方法如图 5.21 所示。所需要做的就是指定该子域的名称服务器，只要提供它的域名和 IP 地址就可以完成这项工作。在为 cs. vu. nl 域中的结点解析名称时，名称解析过程将在某一点上持续进行，方法就是读取 DNS 数据库，该数据库存储在 cs. vu. nl 域的名称服务器上。

名称	记录类型	记录值
cs.vu.nl.	NS	solo.cs.vu.nl.
cs.vu.nl.	NS	star.cs.vu.nl.
cs.vu.nl.	NS	ns.vu.nl.
cs.vu.nl.	NS	top.cs.vu.nl.
ns.vu.nl.	A	130.37.129.4
top.cs.vu.nl.	A	130.37.20.4
solo.cs.vu.nl.	A	130.37.20.5
star.cs.vu.nl.	A	130.37.24.6
star.cs.vu.nl.	A	129.31.231.42

图 5.21　vu. nl 域的部分描述，vu. nl 域包含 cs. vu. nl 域

5.3.4.3　非集中式 DNS 实现

到目前为止我们已介绍的 DNS 实现是一种标准的实现。它遵循的是服务器分层结构，其中有 13 台已知的根服务器，终端是百万台服务器。这里需要注意的重要一点是，高层结点接收的请求比低层结点的要多。只有把高层的名称-地址绑定缓存起来，才有可能避免因发送请求过多而把它们淹没了。

完全非集中式的解决方法就可以避免这些可扩展性问题。我们可以计算出 DNS 名称的散列值，然后把这个散列值作为一个键值，可以在分布式散列表或含有完全划分的根结点的分层定位服务中查找它。这种方法的一个明显缺点是，我们失去了原始名称的结构。这种丢失可能会妨碍在特定域查找所有孩子的有效实现。

另一方面，把 DNS 映射到基于 DHT 的实现有很多优点，最突出的是其可扩展性。正如(Walfish 等 2004)介绍的，当需要多个名称时，使用标识符作为一种与语义无关的方法来访问数据，将使得不同的系统可以使用一个命名系统。其原因很简单：我们已经知道了如何支持(超)大型的(无层次)名称集。所要做的是维护标识符-名称的映射信息，在这种情况下，名称可能是来自 DNS，或者是一个 URL 等。让用户和组织使用严格的本地名称空间，

使得使用标识符更容易。严格的本地名称空间完全类似于在一台计算机上维护一个私有环境变量的设置。

在 CoDoNS(Ramasubramanian 和 Sirer 2004a)中介绍了把 DNS 映射到基于 DHT 的点对点系统。它们使用了一个基于 DHT 的系统，其中，键值的前缀用于路由结点。为了解释方便，请看这样一种情况，标识符的每个数字取自{0，…，b－1}集，其中 b 为基数。例如，在 Chord 系统中，b＝2。如果假设 b＝4，那么来看看标识符为 3210 的结点。假设该在系统中有一个含有如下标识符的结点路由表：

n_0：标识符前缀为 0 的结点

n_1：标识符前缀为 1 的结点

n_2：标识符前缀为 2 的结点

n_{30}：标识符前缀为 30 的结点

n_{31}：标识符前缀为 31 的结点

n_{33}：标识符前缀为 33 的结点

n_{320}：标识符前缀为 320 的结点

n_{322}：标识符前缀为 322 的结点

n_{323}：标识符前缀为 323 的结点

结点 3210 负责处理前缀为 321 的键值。如果它接收了一个键值 3123 的查询请求，将把它转发给结点 n_{31}，结点 n_{31} 又查看是否需要把它转发给前缀为 312 的结点。(注意，每个结点维护另外两个列表，如果其路由表丢失了一项，可以用这两个列表来路由。)该方法的详细内容可以在 Pastry(Rowstron 和 Druschel 2001)和 Tapestry(Zhao 等 2004)中找到。

再来看看 CoDoNS，负责键值 k 的结点存储了 DNS 资源记录，该记录与散列到 k 的域名相关联。但是，其中有趣的部分是，CoDoNS 通过复制资源记录，试图最小化路由一个请求时的跳数。其基本策略很简单：结点 3210 将把其内容复制到前缀为 321 的结点。这种复制使得到结点 3210 的每个路由路径减少一跳。当然，可以对所有前缀为 32 的结点再次使用这种复制，以此类推。

当一个 DNS 记录复制到所有前缀为 i 的结点时，就说是在 i 层被复制。注意，在 i 层被复制的记录(通常)需要 i 个查找步骤。但是，这里需要在复制的层与网络和结点资源的使用之间进行折衷。CoDoNS 这样做导致的汇集查找延时比给定常量 C 小。

更为具体的是考虑一下查询的频率分布性。假设按特定键值被请求的频率来给查找请求排名，被请求最多的键值排在第一位。如果第 n 个排名的频率与 $1/n^{\alpha}$(α 接近 1)成比例，那么就称该查找的分布为 Zipf-like 分布。George Zipf 是哈佛大学的语言学家，他在研究自然语言的词频时发现了这种分布性。但是，事实证明，这种分布性适用于很多其他事情，从城市人口、地震大小、高收入分布、公司收入，甚至 DNS 查询(Jung 等 2002)。

假设 x_i 是在 i 层被复制最多的记录部分，(Ramasubramanian 和 Sirer 2004b)证明，x_i 可以用如下公式表示(这里我们只是告诉读者有这么个公式存在，不去证明它，而是简短地介绍如何使用它)：

$$x_i = \left[\frac{d^i(\log N - C)}{1 + d + \cdots + d^{\log N - 1}}\right]^{\frac{1}{(1-\alpha)}}, \quad 其中 \quad d = b^{(1-\alpha)/\alpha}$$

其中 N 为网络中的结点数，α 是 Zipf 分布中的参数。

该公式使得我们可以决定应复制哪条 DNS 记录。为了说明清楚，来看看 b＝32 和 a＝0.9 的情况。在有 10 000 个结点和 1 000 000 条 DNS 记录的网络中，在进行一个查找时，要实现平均跳数 C＝1，有 x_0＝0.0000701674，即在任何地方只应复制 70 条 DNS 记录。同样，x_1＝0.00330605，在第 1 层只应复制后面的 3306 条记录。当然，这里要求 x_i＜1。在本示例中，x_2＝0.155769，x_3＞1，因此，复制后面的 155 769 条记录，其余记录不复制。平均一跳就足以找到所请求的 DNS 记录。

5.4 基于属性的命名

通常，无层次名称和结构化名称提供了一种唯一的与位置无关的方法来引用实体。而且，结构化名称的设计提供了一种用户友好的方法来命名实体，从而可以很方便地访问这些实体。在大多数情况下，都是假定一个名称只引用一个实体。然而，位置无关和用户友好并不是命名实体的唯一标准。特别是，随着可用的信息越来越多，高效地查找实体已变得很重要了。这种方法要求只需用户提供所有查找实体的一些描述即可。

能提供实体描述的方法有很多，但在分布式系统中，最常用的一种是用(属性，值)来描述实体，通常称为**基于属性的命名**(attribute-based naming)。在这种方法中，假定一个实体有一个相关的属性集。每个属性描述了实体的某个方面。通过指定特定属性应具有的值，用户基本上就可以限定他所感兴趣的实体集。命名系统负责返回满足用户所描述的一个或多个实体。本节将详细介绍基于属性的命名系统。

5.4.1 目录服务

基于属性的命名系统又称为**目录服务**(directory service)，而支持结构化命名的系统通常称为**命名系统**(naming system)。对于目录服务，实体有一个可用于查找的相关属性集。在有些情况下，属性的选择相对简单。例如，在电子邮件系统中，消息可以用发送者、接收者、主题等属性来标记。然而，即使是在电子邮件系统中，当还需要其他类型的描述时，事情会变得很困难，开发邮件过滤器的困难性就说明了这一点。邮件过滤器只允许特定的消息(基于它们的描述)可以通过。

要设计适当的属性集并不是件容易的事情。在大多数情况下，属性设计必须手工完成。即使是大家赞成要使用的某个属性集，实践证明，由不同的人不断地设置属性值本身就是一个问题，很多人在因特网上访问音乐和视频数据库时就有过这种体验。

为了减轻这些问题，研究人员已致力于统一描述资源分方法。在分布式系统中，与此有关的一个开发是**资源描述框架**(resource description framework，RDF)。RDF 模型的基础是，资源用由主题、断言和对象组成的三联体来描述。例如，(Person，name，Alice)描述了资源 Person，它的 name 是 Alice。在 RDF 中，每个主题、断言或对象本身也可以是一个资源。这意味着 Alice 也可以实现为指向一个文件的引用，从而可以被检索。断言则可以包含对该断言的文本描述。当然，与主题和对象有关的资源可以是任何东西。RDF 中的引用基本上是 URL。

如果资源描述已存储，那么就可以以基于属性的命名系统常用的方式来查询。例如，应用程序可以查询与名为 Alice 有关的信息。这种查询将返回一个引用，指向与 Alice 有关的 Person 资源。该资源随后可被应用程序获得。有关 RDF 的详细信息可参见（Manola 和 Miller 2004）。

在这个示例中，资源描述是存储在中心位置。没有理由要求资源都必须存储在相同的位置。但是，没有在相同的地方存储资源描述可能导致一个严重的性能问题。与结构化命名系统不同，在基于属性的命名系统中查找值基本上要求对所有资源描述进行穷举搜索。如果要考虑性能，在单个数据库中进行这种搜索不是问题，但当数据分布在多个可能是很分散的计算机中时，就需要应用单独的技术了。下面我们将介绍在分布式系统中解决这种问题的不同方法。

5.4.2 分层实现：LDAP

处理分布式目录服务的常用方法是把结构化命名与基于属性的命名合起来使用。这种方法已被广泛采用，例如，Microsoft 的活动目录服务和其他系统。这些系统很多使用或依赖的是**轻量级目录访问协议**（lightweight directory access protocol，LDAP）。LDAP 目录服务是从 OSI 的 X.500 目录服务派生而来的。尽管 OSI 的服务很多，但相关实现的质量阻碍了广泛的使用，要使得它可用，需要进行简化。有关 LDAP 的详细信息可以参见（Arkills 2003）。

从概率上说，LDAP 目录服务由多个记录组成，这些记录通常称为目录项。目录项与 DNS 中的资源记录类似。每个记录由（属性，值）对集组成，每个属性有一个相关的类型。单值属性与多值属性之间有明显的不同。多值属性通常表示数组和列表。图 5.22 显示了一个简单的目录项，标识了图 5.20 中的服务器的网络地址。

属性	缩写	值
Country	C	NL
Locality	L	Amsterdam
Organization	O	Vrije University
OrganizationalUnit	OU	Comp. Sc.
CommonName	CN	Main Server
Mail_Servers	—	137.37.20.3, 130.37.24.6, 137.37.20.10
FTP_Server	—	137.37.20.20
WWW_Server	—	137.37.20.20

图 5.22 一个简单的使用了 LDAP 命名规范的 LDAP 目录项示例

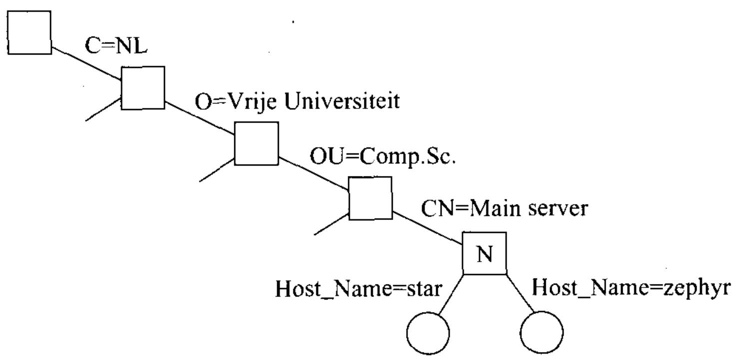
在这个示例的前 5 个属性中，使用了 LDAP 标准中介绍的命名规范。Organization 和 OrganizationalUnit 分别描述了与存储在记录中的数据有关的组织和部门。同样，属性 Locality 和 Country 提供了该项存储位置的额外信息。属性 CommonName 通常用作一个（模糊的）名称，以标识目录中有限部分的一项。例如，如果给定了其他 4 个属性 Country、Locality、Organization 和 OrganizationalUnit 的具体值，名称 Main Server 就足以找到相应的项。在这个示例中，只有属性 Mail_Servers 有与之相关的多个值。其余所有属性都只有一个值。

在 LDAP 目录服务中，所有目录项的集合称为**目录信息库**(directory information base，DIB)。DIB 的一个重要方面是，每个记录都是唯一命名的，从而可以被查找。这种全局唯一的名称在每个记录中是以一个命名属性系列出现的。每个命名属性称为**相关识别名**(relative distinguished name，RDN)。在图 5.22 的示例中，前 5 个属性都是命名属性。如图 5.22 所示，使用 LDAP 中表示命名属性的规范缩写，属性 Country、Organization 和 OrganizationalUnit 可用来构成全局唯一名：

```
/C=NL/O=Vrije University/OU=Comp. Sc.
```

它类似于 DNS 名 nl. vu. cs。

就像在 DNS 一样，通过依次列出 RDN 来使用全局唯一名称，使得目录项集合形成了分层，称之为**目录信息树**(directory information tree，DIT)。DIT 基本上就构成了 LDAP 目录服务的命名图，其中，每个结点表示一个目录项。另外，结点还可以起传统意义上的目录的作用，其中，作为父结点的结点可以有多个孩子结点。请看图 5.23(a)所示的部分命名图。



(a)

属性	值
Country	NL
Locality	Amsterdam
Organization	Vrije Universiteit
OrganizationalUnit	Comp.Sc.
CommonName	Main server
Host_Name	star
Host_Address	192.31.231.42

属性	值
Country	NL
Locality	Amsterdam
Organization	Vrije Universiteit
OrganizationalUnit	Comp.Sc.
CommonName	Main server
Host_Name	zephyr
Host_Address	137.37.20.10

(b)

图　5.23

(a) 目录信息树的一部分；(b) 两个目录项，以 Host_Name 作为 RDN

结点 N 对应于图 5.22 所示的目录项。该结点同时又作为其他多个目录项父结点，这些目录项具有另外一个属性 Host_Name，该属性用作 RDN。例如，这些项可以用来表示图 5.23(b)所示的主机。

因此，正如我们前面介绍的那样，LDAP 命名图中的结点可以同时表示传统意义上的目录和 LDAP 记录。这是由两种不同的查找操作来支持的。给定 DIT 的路径名，read 操作可用于读取一个记录。而 list 操作可用于列举出 DIT 中给定结点的所有输出边的名称。每

个名称对应着给定结点的一个孩子结点。注意，list 操作不返回任何记录，它只返回名称。换句话说，当输入如下名来调用 read 时，将返回图 5.22 所示的记录：

```
C/=NL/O=Vrije University/OU=Comp.Sc./CN=Main server
```

而调用 list 则从图 5.23(b)所示的项中返回名称 star 和 zephyr，以及其他以类似方式注册的主机名称。

实现 LDAP 目录服务进程的方法与实现诸如 DNS 之类的命名服务相同，只不过 LDAP 支持更多的查找操作，我们稍后将讨论。当处理大型目录时，通常把 DIT 进行分隔并分布在多个称为**目录服务代理**(directory service agent，DSA)的服务器上。这样，已分隔的 DIT 的每个部分对应于 DNS 的一个区域。同样，每个 DSA 的行为很像通常的名称服务器，只不过它实现了一些常见的目录服务，如高级搜索操作。

客户由**目录用户代理**(directory user agent，DUA)表示。DUA 类似于结构化命名服务中的名称解析器。DUA 按照标准化的访问协议与 DSA 进行信息交换。

LDAP 实现与 DNS 实现不同的是那些通过 DIB 进行搜索的工具。尤其是，如果给定了要搜索项的属性应满足的标准，它就可搜索目录项。例如，假设要列举出 Vrije 大学所有的主服务器。使用(Howes 1997)中定义的表示法，那么利用如下搜索操作就可以返回一个列表：

```
answer=search("&(C=NL)(O=Vrije University)(OU=*)(CN=Main server)")
```

在这个示例中，我们指定了查找主服务器的地方是在 NL 国家的名为 Vrije University 的组织，但对特定的组织单元不感兴趣。但是，每个返回结果的 CN 属性应等于 Main server。

正如我们已经介绍的那样，在目录服务中搜索通常是一个开销较大的操作。例如，为找到 Vrije University 的所有主服务器，要求搜索每个部门的所有项，并把这些结果合并到一个答案中。换句话说，我们通常需要访问 DIT 的多个叶结点以便得到答案。在实践中，这还意味着需要访问多个 DSA。而命名服务往往可以以这样的一种方式来实现：查找操作要求访问的只是单个叶结点。

通过允许多棵树共存并相互链接，整个 LDAP 的设置就可再进一步了。在 Microsoft 的活动目录中就使用了这种方法，形成了一个 LDAP 森林(Allen 和 Lowe-Norris 2003)。显然，在这种组织结构中搜索会非常复杂。为了回避一些可扩展性问题，活动目录往往假设有一个全局索引服务器(称为全局种类)，可以首先进行搜索。该索引表明了下一步需要搜索的 LDAP 域。

尽管为了扩展性，LDAP 本身已开发为分层结构，但通常会把 LDAP 和 DNS 组合起来。例如，LDAP 的每棵树需要根结点(在活动目录中称为域控制器)是可访问的。根结点通常是用一个 DNS 名称进行命名的。这就可以通过一个适当的 SRV 记录来找到，我们前面已经介绍过。

LDAP 通常表示一种支持基于属性命名的标准方法。另外还开发了其他一些目录服务，它们使用的是更为传统的方法，最明显的是在网格计算和 Web 服务环境中。一个示例是**统一目录与发现集成**(universal directory and discovery integration，UDDI)。

这些服务假设在实现中，一个或只有少数几个结点相互协作以维护分布式数据库。从

技术的角度来看，这没什么新鲜的。同样，当阅读 UDDI 几百页的说明时，也没有发现什么新东西(Clement 等 2004)。基本方案总是相同的：通过使多个数据库对应用程序是可访问的，然后这些应用程序负责分别查询每个数据库并汇集查询结果，就可以实现可扩展性。中间件支持也是这样。

5.4.3 非集中式实现

随着点对点系统的出现，研究人员也已经为非集中式的、基于属性的命名系统寻找解决方法。这里的关键问题是，(属性，值)对需要有效地进行映射，以便搜索能高效地完成，即是避免在整个属性空间进行穷举搜索。下面我们来看看实现这种映射的几种方法。

5.4.3.1 映射到分布式散列表

我们先来看看(属性，值)对需要基于 DHT 系统的支持的情况。首先，假设查询是由属性对的联合与 LDAP 组成的，也就是说，用户指定了一个属性列表以及他要查找的各个属性所具有的唯一值。这种查询最大的优点是不需要指定范围。范围查询会大大地增加把属性对映射到 DHT 的复杂性。

在 INS/Twine 系统中支持单值查询(Balazinska 等 2002)。假设每个实体(称为一个资源)可以用如图 5.24 所示的分层组织的属性来描述。每个这样的描述可以转换为**属性-值树**(attribute-value tree，AVTree)，可以把它作为映射到基于 DHT 的系统的编码基础。

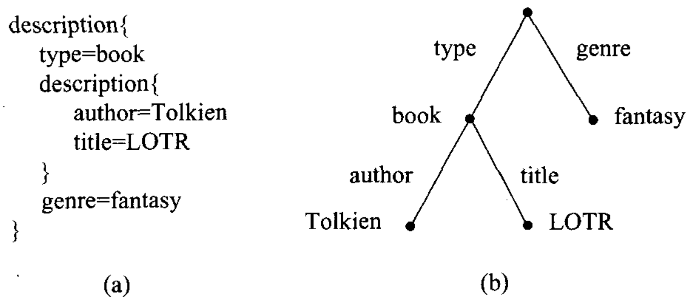


图 5.24

(a) 资源的一般描述；(b) 表示为 AVTree

这里的主要问题是把 AVTree 转换为键值集，这些键值可以在 DHT 系统中进行查找。在这种情况下，每条起始于根结点的路径被赋给了一个唯一的散列值，其中，路径描述以一个链接(表示一个属性)开头，以一个结点(值)或另一个链接结尾。看看图 5.24(b)，所有路径的散列如下所示：

h_1：hash(type-book)

h_2：hash(type-book-author)

h_3：hash(type-book-author-Tolkien)

h_4：hash(type-book-title)

h_5：hash(type-book-title-LOTR)

h_6：hash(genre-fantasy)

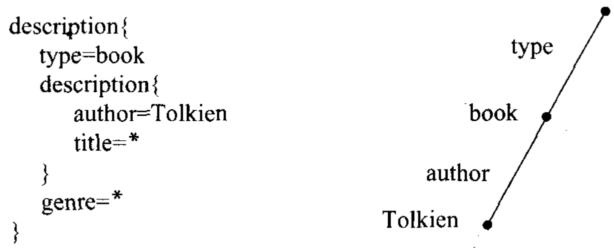
具有散列值 h_i 的结点保存实际的资源(或指向它的引用)。在本示例中，有 6 个结点存储了 Tolkien 的 Lord of the Rings 一书的信息。这种冗余的好处是，它允许支持部分查询。

例如，考虑这样一个查询："返回由 Tolkien 编著的所有图书"。该查询转换为图 5.25 所示的 AVTree，要计算如下 3 个散列值：

h_1：hash(type-book)

h_2：hash(type-book-author)

h_3：hash(type-book-author-Tolkien)



(a) (b)

图 5.25

(a) 查询的资源描述；(b) 表示为 AVTree

这些值被发送给存储了 Tolkien 的图书的结点，并至少可以返回 Lord of the Rings 一书。注意，诸如 h_1 的散列值很常见，会经常生成。这些类型的散列值可以过滤出系统之外。而且，不难看出，只有最特殊的散列值才需要计算。详细信息请参见(Balzinska 等 2002)。

现在，让我们来看看另一种类型的查询，即那些可以包含属性值的区间说明的查询。例如，要查找房屋的某人，通常希望指定；其价格必须位于特定的区间内。同样有几种解决方法，在第 13 章讨论发布/订阅系统时我们会遇到其中一些。这里我们来讨论 SWORD 资源发现系统中采用的一种解决方法(Oppenheimer 等 2005)。

在 SWORD 系统中，由资源描述给出的(属性，值)对首先转换为 DHT 的一个键值。注意，这些属性值对总是含有一个值，只有查询才可能含有属性的区间值。当计算散列值时，属性的名称与其值是分开的。换句话说，结果键值中的某些位表示属性名，而其他位表示其值。而且，该键值含有一些随机位，以保证要生成的所有键值的唯一性。

这样，就可以很方便地划分属性空间：如果 n 位保留为对属性名编码，就有 2^n 种不同的服务器组可用，每组一个属性名。同样，使用 m 位来对值编码，已进一步划分的每个服务器组就可用来存储特定(属性，值)对。DHT 只用作分布式属性名。

对每个属性名，其值的可能区间被划分为子区间，每个服务器被赋给一个子区间。为了解释方便，请看一个含有两个属性的资源描述：a_1 的值为区间[1…10]，a_2 的值为区间[101…200]。假设对于 a_1 有两个服务器：s_{11} 负责记录 a_1 在区间[1…5]的值，s_{12} 负责记录 a_1 在区间[6…10]的值。同样，s_{21} 负责记录 a_2 在区间[101…150]的值，s_{22} 负责记录 a_2 在区间[151…200]的值。于是，当资源获得值($a_1=7$，$a_2=175$)，那么就必须通知服务器 s_{12} 和 s_{22}。

这种方案的优点是，可以很容易地支持区间查询。当某个查询请求返回位于 165～189 的 a_2 时，该请求被转发到服务器 s_{22}，该服务器将返回与该区间匹配的资源。但是，其缺点是，更新信息需要发送给多个服务器。而且，不能立即看出各个服务器之间的负载平衡情况。尤其是当某个区间的查询特别频繁时，某个服务器所接收的查询就会占总查询的很大部分。如何解决基于 DHT 系统的这种负载平衡问题可参见(Bharambe 等 2004)。

5.4.3.2 语义覆盖网络

基于属性的命名系统的非集中式实现已经显示了各种结点的自治度是在不断地增加

的。与诸如分布式 LDAP 的系统相比,基于属性的命名系统对结点的加入和离开更不那么敏感。当结点具有资源描述从而可以被其他结点发现时,其自治度将进一步增加。换句话说,(属性,值)对在结点集中传播并没有预定的方案。

由于没有这类方案,使得结点必须去发现被请求的资源在哪里。这种发现对非结构化的覆盖网络是很典型的,我们已经在第 2 章进行了介绍。为了使搜索有效,很重要的一点是,一个结点具有指向其他结点的引用,而这些结点最可能就是查询的答案。假设起始于结点 P 的查询与 P 具有的资源关系密切,那么我们要寻求的是为 P 提供一个指向语义近似的邻结点的链接集。回忆一下可以知道,这种列表又称为**局部试图**(partial view)。语义近似可以以不同的方式来定义,但归根结底都是要与具有类似资源的结点保持联系。结点和这种链接就构成了所谓的**语义覆盖网络**(semantic overlay network)。

语义覆盖网络的一种常用方法是假设在每个结点维护的元信息具有共性。换句话说,存储在每个结点中的资源使用的是相同的属性集(更准确地说是相同的数据方案)(Crespo 和 Garcia-Molina 2003)。有了这种方案,就可以在结点之间定义很类似的函数。这样,每个结点就只存储到最类似的邻结点的链接,并且在查找特定数据时首先查询这些结点。注意,只有当我们可以假设由某个结点发起的查询与存储在该结点的内容有关系时,这种方法才有意义。

但是,假定数据方案有共性往往是错误的。在实践中,在不同结点之间,资源中的元信息是很不一致的,要达成一致以及用什么如何描述资源几乎是不可能的。由于这个原因,语义覆盖网络通常需要寻找不同的方法来定义类似性。

一种方法是完全忘记掉属性,只考虑很简单的描述,如文件名。通过跟踪哪些结点肯定会响应文件搜索,就可以完成一个覆盖网络的被动构建。例如,(Sripanidkulchai 等 2003)首先发送一个查询给结点的语义邻结点,但如果被请求的文件不在这里,就要进行(有限的)广播。当然,这种广播可能会导致语义结点列表的更新。注意,有意义的是,如果一个结点请求其语义邻结点转发一个查询给它们的语义邻结点,其影响是最小的(Handrukande 等 2004)。这种现象可以用所谓的**小范围影响**(small-world effect)来解释。小范围影响表示,Alice 的朋友也是其他人的朋友(Watts 1999)。

构建语义邻结点列表的一种更前摄的方法是由(Voulgaris 和 van Steen 2005)提出的,它使用了一个**语义近似函数**(semantic proximity function),该函数定义在两个结点 P 和 Q 的文件列表 FL_P和 FL_Q之上。该函数计算 FL_P和 FL_Q中共有文件的数量。然后,其目标是,通过让某个结点只保持邻结点(这些邻结点与该结点具有的共有文件最多)的列表来优化这个近似函数。

最后,图 5.26 部署了一个两层的 gossiping 方案。底层由一个感染协议组成。感染协议的目的是维护统一的随机选择结点的部分视图。正如在第 2 章解释的那样,有多种不同的方法来实现这点(Jelasity 等 2005a)。顶层通过 gossiping 维护一个语义近似邻结点的列表。要开始一个交换,结点 P 可以从其当前列表中随机选择一个邻结点 Q,但这里的技巧是,让 P 只发送那些语义上与 Q 最接近的项。反过来当 P 接收来自 Q 的项时,它最后会保持一个只是由语义最近似的结点组成的部分视图。事实证明,由顶层维护的部分视图将快速地靠近最优。

至此,很明显,语义覆盖网络与非集中式搜索关系最为接近。在(Risson 和 Moors

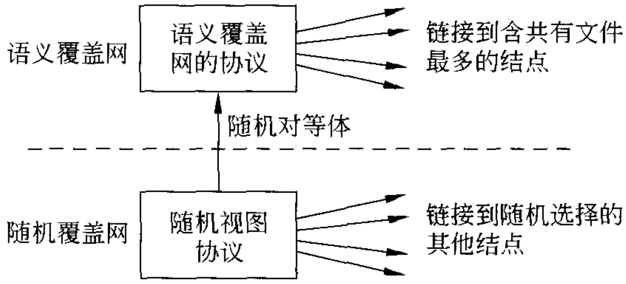


图 5.26 通过 gossiping 维护一个语义覆盖网

2006)中，介绍了在所有点对点系统中进行搜索的扩展概述。

本章小结

名称可用来表示实体。基本上有 3 种类型的名称。地址是与实体相关的访问点的名称，也可以简称为实体地址。标识符是另一种类型的名称。标识符有三个属性：每个实体都只有被一个标识符引用；一个标识符只能引用一个实体，不能再赋给其他实体。最后，用户友好的名称的目标是由人来使用，因此由字符串表示。

无层次命名系统需要把标识符解析为其相关实体的地址。实体的这种定位可以以不同的方法来完成。第 1 种方法是使用广播或多播。实体的标识符被广播给分布式系统的每个进程。为实体提供访问点的进程通过提供访问点的地址进行响应。很明显，这种方法的可扩展性有限。

第 2 种方法是使用转发指针。每当一个实体转移到另一个位置时，它就会留下一个指针，说明它下一步所在的位置。定位实体需要遍历转发指针形成的路径。为了避免形成太长的指针链，定期缩短指针链是很重要的。

第 3 种方法就是给实体指定一个宿主位置。每当实体转移到另一个地方时，它都会通知宿主位置，告诉宿主位置自己当前的位置。在定位实体时，首先询问宿主位置，以便了解实体的当前位置。

第 4 种方法是把所有结点组织成一个结构化的点对点系统，同时考虑到各个实体的标识符，把结点赋给实体。随后，通过设计一个路由算法，把查询请求移到指定实体的结点，从而可能实现高效且健壮的名称解析方法。

第 5 种方法是创建一棵分层搜索树。网络划分成不重叠的域。域可以组合成更高层的(不重叠)域，以此类推。这里只有一个顶层域，它涵盖了整个网络。每个层次的每个域都有关联的目录结点。如果一个实体位于域 D 中，那么更高一层的域的目录结点将拥有一个指向 D 的指针。最低层目录结点存储了实体的地址。最高层的目录结点知道所有的实体。

结构化名称可以很容易地组织成一个名称空间。名称空间可以用一个命名图来表示，在这个命名图中，一个结点表示的是一个具名实体，而边的标识则表示该名称位于哪个实体下。具有多条进入边的结点表示的是一个实体集，也可以是一个上下文结点或目录。大型的命名图往往组织成基于根结点的无环有向图。

命名图可以以结构化的方式很方便地组织用户友好的名称。一个实体可以用一个路径名来引用。名称解析的过程是，通过查找路径名的各个组成部分(每次一个)来贯穿该命名图。大型的命名图是通过把结点分布在多个名称服务上来实现的。当通过贯穿命名图来解

析路径名时，一旦某个结点到达实现了它的名称服务器，名称解析就在下一个名称服务器中继续进行。

基于属性的命名方法的问题更多。在基于属性的命名系统中，实体是由(属性，值)对集来描述的。查询也是以这样的对来表达的，它基本上都要求对所有标识符进行穷举搜索。只有当这些标识符都存储在单个数据库中时，这种搜索才是可行的。人们还设计了其他的解决方法，把属性值对映射到基于 DHT 的系统，使得实体标识符集可以实现分布性。

利用分布式搜索技术，基于属性的命名系统逐步取代了名称解析。在语义覆盖网络中就使用了这种方法。在语义覆盖网络中，结点维护一个有关其他结点的本地列表，这些结点在语义上具有类似的内容。通过首先查询紧接着的邻结点，只有在此没有找到后继者之后才部署(有限的)广播，这些语义列表就可以实现高效搜索。

习　题

1. 举出一个例子，在这个例子中，为了真正访问实体 E，需要把它的地址进一步解析成另一个地址。
2. 你认为像 http://www.acme.org/index.html 这样的 URL 是不是与位置无关的? http://www.acme.nl/index.html 呢?
3. 请给出一些正确标识符。
4. 是否允许标识符包含有关它所引用的实体的信息?
5. 简要描绘一下全局唯一标识符的一个有效实现。
6. 考虑图 5.4 所示的 Chord 系统，假设结点 7 刚刚加入该网络中。其指状表会是什么样子? 其他指状表需要做什么修改吗?
7. 考虑一个基于 DHT 的 Chord 系统，其中，m 位标识符空间的 k 位保留给了超级体。如果标识符是随机赋值的，在有 N 个结点的系统中，可能有多少个超级体?
8. 如果往 Chord 系统中插入了一个结点，是否需要立即更新所有指状表?
9. 当要在基于 DHT 的系统中解析一个键值时，递归查询的主要缺点是什么?
10. 有一种定位实体的特殊方式，称为 anycasting，通过这种方式，可以使用 IP 地址来标识服务(请参见 RFC 1546)。向 anycast 地址发送请求时，实现了该 anycast 地址所标识服务的服务器会返回一条响应消息。简要说明如何实现基于分层定位服务(在 5.2.4 节介绍)的 anycast 服务。
11. 两层的基于宿主的方法是分层定位服务的一种特殊形式，请问其根在哪里?
12. 假设已知某个移动实体几乎从不会离开域 D，即使真的离开了，它也会很快返回。如何使用这条信息在分层定位服务中加快查询操作的速度?
13. 在深度为 k 的分层定位服务中，当移动实体改变它的位置时，最多需要更新多少条位置记录?
14. 假设一个实体从位置 A 转移到 B，期间经过了几个中间位置，在每个中间位置都只停留相对较短的时间。到达 B 后，它停留了一段时间。在分层定位服务中更改地址可能还需要相对较长的时间，因此在访问中间位置时应该避免更改地址。那么，当实体位于中间位置时应该如何查找它?
15. 分层定位服务中的根结点可能是一个潜在的瓶颈。如何能有效地避免这个问题?

16. 请举例说明 URL 闭合机制的工作方式。
17. 请说明 UNIX 系统中硬链接与软链接之间的区别。是否有某些事情硬链接能做而软链接不能？反过来呢？
18. DNS 中的高层名称服务器(就是那些在 DNS 名称空间中接近根的、实现了结点的名称服务器)一般不支持递归名称解析。如果让它们支持递归名称解析，那么是否有大幅度的性能改善？
19. 请解释如何使用 DNS 来实现基于宿主位置的方法以定位移动主机。
20. 在大多数的 UNIX 系统中是如何查找挂载点的？
21. 考虑一个使用了用户名空间的分布式系统，每个用户都有自己私有的名称空间。来自这种名称空间的名称是否可用于在两个不同用户之间共享资源？
22. 考虑一下 DNS。要引用子域(该域实现为一个不同的区而不是当前域)中的结点 N，需要为该区指定一个名称服务器。这个服务器地址是否总是需要包含一个资源记录，还是有时候只提供其域名即可？
23. 挂载共有文件是定义语义近似的幼稚方法。假设你想基于文本文件来构建语义覆盖网络，你还能想到其他哪些语义近似函数？
24. (**实验**)设置你自己的 DNS 服务器。在 Windows 或 UNIX 机器上安装 BIND 软件，并为一些简单的名称配置它。使用诸如 Domain Information Groper(DIG)的软件来测试你的配置。应确保你的 DNS 数据库包含有名称服务器、邮件服务器和标准服务器的记录。注意，如果你是在名为 HOSTNAME 的机器上运行 BIND，就应能解析形如 RESOURCE-NAME. HOSTNAME 的名称。

第6章

同步化

在前面的几章中，我们已经讨论了进程及进程间的通信。虽然进程间通信很重要，但它并不是分布式系统的全部内容。与之紧密相关的问题是进程间如何协作和同步。进程协作通过命名方式得到部分支持，进程协作使得进程可以共享资源，或者一般地说是共享实体。

本章中，我们主要集中精力研究进程间是如何同步的。例如，多个进程不能同时访问一个共享资源(例如打印机)，而是相互协作，彼此授权暂时地独占访问。另一个例子是多个进程有时可能需要就事件的顺序达成一致，比方说来自进程 P 的消息 m1 是在来自进程 Q 的消息 m2 之前还是之后被发送出去。

实践表明，分布式系统的同步常常比单处理器或者多处理器系统中的同步更加困难。本章中讨论的问题及其解决方法本质上是比较常见的，它们出现在分布式系统的许多不同情况下。

我们从基于真实时间的同步问题开始讨论，接着讨论以相对排序的方式而不是以绝对时间的方式来实现的同步。我们也要讨论分布式全局状态的概念，以及如何通过进程同步来记录全局状态。

在许多情况下，一组进程能指定一个进程作为协作者是重要的，这可以通过选举算法来实现。我们将在单独的一节中讨论各种各样的选举算法。

与同步相关的两个主题是分布式系统中的互斥和分布式事务。分布式互斥保护共享资源不被多个进程同时访问。分布式事务也作类似的事情，但是它通过高级的并发控制机制来优化访问。互斥和事务将在单独的小节中讨论。

许多不同的分布式系统有多种风格迥异的分布式算法。许多例子(以及更多的参考资料)可以在(Andrews 2000;Guerraoui 和 Rodrigues 2006)中找到。大量算法的更为正式的方法可以在(Attiya 和 Welch 2004;Lynch 1996 和 Tel 2000)中找到。

6.1 时钟同步

在集中式系统中，时间是明确的。当进程想知道时间时，它就进行一次系统调用，然后系统内核就会告诉它。如果进程 A 询问时间，稍后进程 B 也询问时间，那么进程 B 获取的时间值将比进程 A 获取的时间值大，或者可能相等，但无论如何不会小于后者。在分布式系统中，取得时间上的一致是不容易的。

作为一个例子，考虑 UNIX 中的 make 程序在缺少全局时间时的情况。通常在 UNIX 系统中会把较大的程序分成多个源文件，当一个源文件发生变化时，只需重新编译一个文件即可，不需要对所有的文件进行重新编译。如果一个程序包含 100 个文件，在修改一个文件时不需要重新编译所有的文件，这将大大提高编程人员的工作效率。

通常 make 程序工作的方式很简单。程序员修改完了所有的源文件后启动 make 程序，make 程序检查所有的源文件和目标文件的最后修改时间。如果源文件 input. c 的时间是 2151，而相应的目标文件 input. o 的时间是 2150，那么 make 程序知道在 input. o 创建之后曾经对 input. c 进行过修改，因此必须对 input. c 进行重新编译。相反，如果 output. c 的时间是 2144，而 ouput. o 的时间是 2145，那么这种情况下就不需要重新编译。make 程序就是这样检查所有的源文件，找出需要重新编译的，然后调用编译器重新编译它们。

现在设想一下在没有全局统一时间的分布式系统中会发生什么。假设 output. o 的时间如上为 2144，紧接着 output. c 被修改，但是由于它所在机器上的时钟略慢，结果 output. c 的时间值被赋为 2143，如图 6.1 所示。make 程序将不调用编译器。最终的可执行的二进制程序将包含由旧的源文件和新的源文件所产生的混合目标文件。该程序可能崩溃，而程序员很难找出错误出在哪里。

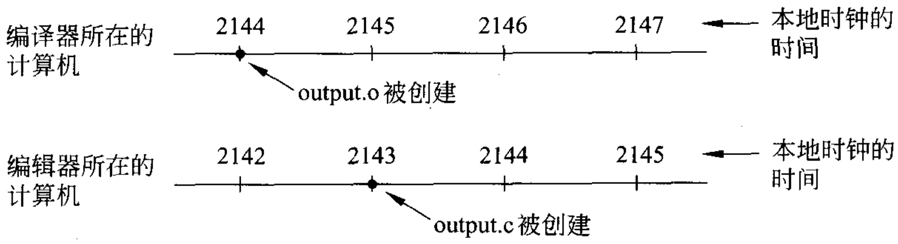


图 6.1 当每台机器都有它自己的时钟时，一个发生于另一个事件之后的事件可能会被标记上较早的时间

由于时间是人们思考问题的基础，并且我们已经看到，所有的时钟没有完全同步的后果是如此可怕，所以我们先从一个简单问题开始有关同步的学习：是否有可能对分布式系统中的所有时钟进行同步？

6.1.1 物理时钟

几乎所有的计算机都有一个计时电路。尽管一般使用“时钟”这个词来表达这些设备，但它们实际上不是通常意义的时钟，把它们称为**计时器**（timer）可能更恰当一点。计算机的计时器通常是一个精密加工过的石英晶体。石英晶体在其张力限度内以一定的频率振荡，这个频率取决于晶体本身如何被切割及其受到张力的大小。有两个寄存器与每个石英晶体相关联，一个是**计数器**（counter），另一个是**保持寄存器**（holding register）。石英晶体的每次振荡使计数器减 1。当计数器减为 0 时，产生一个中断，计数器从保持计数器中重新装入初始值。这种方法使得对一个计时器进行编程，令其每秒产生 60 次中断（或者以任何其他希望的频率产生中断）成为可能。每次中断称为一个**时钟滴答**（clock tick）。

系统初次启动时，通常要求用户输入日期和时间，然后将它们转换成某一个已知起始时间后的时钟滴答次数，并将它存储在存储器中。许多计算机都有一个用特殊的电池支持的 CMOS RAM，其目的是为了以后启动时不再需要输入日期和时间。时钟每滴答一次，中断

服务程序就使存储在存储器里的时间值加1。用这种方法进行(软)时钟计时。

在单机单时钟的情况下,如果这个时钟有少许偏差是不会有多大问题的。因为这台机器上的所有进程使用同一个时钟,所以它们内部仍然会保持一致。例如,如果文件 input.c 的时间是2151,文件 input.o 的时间为2150,那么 make 程序将重新编译源文件。即使时钟偏差了2个单位,即两个文件的真实时间应该分别为2153和2152,make 程序仍然会重新编译源文件。真正重要的是相对时间。

一旦引入多CPU系统,每个CPU都有自己的时钟,情况将发生变化。尽管石英晶体振荡的频率通常是相当稳定的,但仍不可能保证不同计算机中的石英晶体都以完全相同的频率在振荡。实际上,当一个系统有n台计算机时,所有n个晶体都将以略微不同的速度振荡,导致(软)时钟逐渐不同步;当同时读这些时钟值时,将得到不同的值。这种时间值的不同称为**时钟偏移**(clock skew)。时钟偏移的后果如在上面所举的有关 make 的例子中所看到的那样,那些期望与文件、对象、进程或消息相关的时间(即所使用的时钟)不仅正确,而且独立于产生它们的机器的程序可能失败。

在一些系统(例如实时系统)中,真实的时钟时间是很重要的。对于这些系统,外部物理时钟是必须的。考虑到效率和冗余,通常认为使用多个物理时钟比较合适,但有两个问题:(1)我们如何使它们与真实世界的时钟同步呢?(2)我们如何使这些时钟彼此同步?

在回答这些问题之前,让我们稍稍离题来介绍在实际中是如何测量时间的。它不像人们想象的那样简单,尤其是当精度要求很高时更是如此。自从17世纪机械钟发明以来,人们就一直用天文学的方法测量时间。每天太阳都是从东方地平线上升起,然后升到天空的最高处,最后再落到西边。太阳到达天空中它出现的最高点时称为**中天**(transit of the sun),它在每天正午发生。两次连续的太阳中天之间的时间称为一个**太阳日**(solar day)。因为每天有24h(小时),每小时有3600s,所以一个**太阳秒**(solar second)被精确地定义为1/86400个太阳日。平均太阳日的几何算法如图6.2所示。

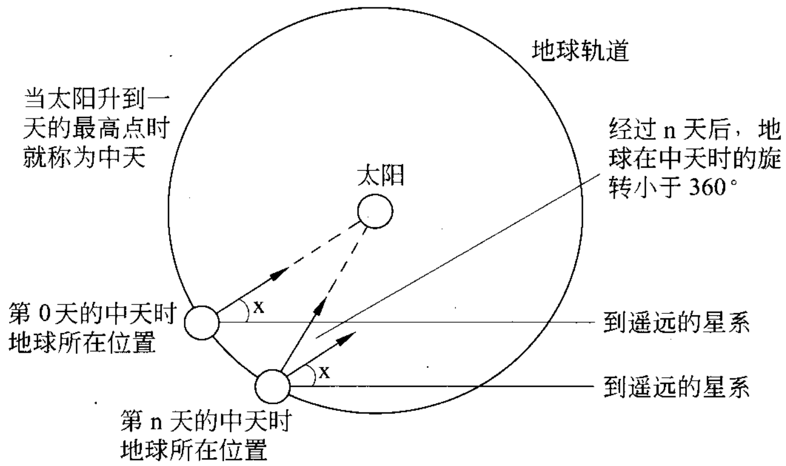


图6.2 平均太阳日的计算

在20世纪40年代,科学家们证实了地球的自转周期并非常数。由于潮汐摩擦和大气的阻力,地球自转速度正在变慢。基于对远古时代珊瑚的生长图案的研究,地质学家现在相信,在3亿年前每年大约有400天。年的长度(地球绕太阳一周的时间)被认为是不变的,那么每天的时间就简单地变长了。除了这种长期变化趋势,也存在一天长短的短期变化,这可

能是由地球地核层熔岩的剧烈沸腾引起的。这些发现促使天文学家们在计算天的长度时测量很多天的长度，然后对它们取平均值；最后除以86 400得到的结果称为**平均太阳秒**(mean solar second)。

1948年原子时钟诞生，它使更精确地测量时间成为可能。原子时钟不受地球的摆动和振动的影响，而是通过铯133原子的跃迁计时。物理学家从天文学家的手中接管了计时的工作，定义1秒是铯133原子作9 192 631 770次跃迁所用的时间。选择9 192 631 770是为了使原子秒与引入原子秒那一年的平均太阳秒相等。目前，世界上大约有50个实验室拥有铯133时钟。每个实验室都定期向巴黎的Bureau International de l'Heure(BIH)报告其时钟的滴答次数。BIH将这些值平均起来产生**国际原子时间**(international atomic time, TAI)。这样TAI就是铯133时钟从1958年1月1日午夜(起始时间)以来被9 192 631 770除后的平均滴答数。

尽管TAI相当稳定，并且任何人只要愿意都可以买到一只铯时钟，但是它仍然存在一个严重的问题，那就是86 400个TAI秒现在比一个平均太阳日少3μs(因为平均太阳日越来越长)。使用TAI计时将意味着多年以后，中午会出现得越来越早，直到最终出现在凌晨。人们也许会注意到这一点，今后可能将会发生与1582年古罗马教皇Pope Gregory十三世宣布从日历中删除10天时类似的情况。这一事件导致了街头暴动，因为地主要收一个整月的地租，银行家要收一个整月的利息，而雇主拒绝向雇员支付他们没有工作的那10天的工资，这里只是提到了其中的几个冲突。在一些新教区，人们拒绝任何与教皇法令有关的事情，有长达170年不接受罗马教皇颁布的日历。

BIH通过引入**闰秒**(leap second)解决了该问题，即当TAI和太阳秒计时之间的差增加到800微秒时使用一次闰秒。闰秒的使用如图6.3所示。这种修正产生了一种时间系统，该时间系统基于恒定长度的TAI秒，但是却和太阳的运动保持一致；它被称为**统一协调时间**(universal coordinated time, UTC)。UTC是所有现代人计时的基础。它从根本上取代了原有的标准，即**格林尼治平均时间**(greenwich mean time)，后者是一种天文时间。

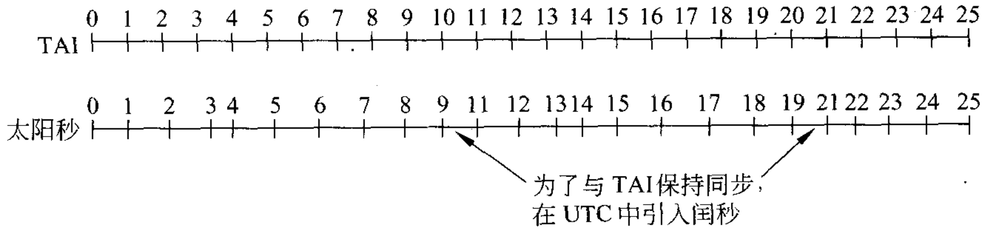


图6.3 TAI秒长度恒定，与太阳秒不同。当需要与太阳保持一致时引入闰秒

大多数电力公司将他们的60Hz或者50Hz时钟的计时基础放在了UTC上。因此当BIH宣布闰秒后，电力公司将他们使用的频率分别从60Hz或者50Hz增加到61Hz或者51Hz，以便于把他们分布地区的时钟向前拨。因为对于计算机来说1秒是一个相当大的时间间隔，所以需要在几年间保持精确时间的操作系统必须有专门的软件根据闰秒的定义来计算闰秒(除非使用电力线的频率来计时，而这种计时方法通常很粗糙)。到目前为止，使用闰秒的次数大约是30次。

为了向需要精确时间的人们提供UTC，国际时间标准化组织(National Institute of Standard Time, NIST)使用了一个位于科罗拉多州柯林斯堡的称为WWV的短波广播站。

WWV 在每个 UTC 秒开始时广播一个小脉冲。WWV 本身的准确度为±1ms，但由于随机空气波动能影响信号传播路径的长度，实际的精确度仅能达到±10ms。英格兰的 MSF 站(位于沃里克郡的拉格比)也提供类似服务，其他一些国家也有类似的站点。

一些地球卫星也提供 UTC 服务。**同步环境操作卫星**(geostationary environment operational satellite)能提供精确到 0.5ms 的 UTC，其他一些卫星提供的精度甚至更高。

使用短波广播或者卫星服务需要准确地了解发送信号和接收者的相对位置，以便对信号传输延迟进行补偿。接收 WWV、GEOS 和其他 UTC 信号的无线电接收器在市场上可以买到。

6.1.2 全球定位系统

在开始真正的时钟同步问题之前，我们首先来看看一个相关的问题，即确定某人在地面上的地理位置。这个定位问题可以通过一个高度专业的分布系统——**GPS**(为**全球定位系统**(global positioning system)的缩写)来解决。GPS 是一个基于卫星的分布式系统，于 1978 年发射。尽管过去它主要是用于军事目的，但近年来发现了很多民用应用，比较著名的就是交通导航。其实还有很多应用领域存在。例如，现在的 GPS 电话可以使呼叫者跟踪其他每个人的位置，当你迷路或有困难时，这个特性非常有用。这同样也可以很容易地应用到跟踪其他东西，包括宠物、小孩、汽车、轮船等等。Zogg(2002)给出了 GPS 的很好概述。

GPS 使用了 29 颗卫星，每个卫星在高度约为 20 000km 的轨道上运行。每个卫星使用了多达 4 个原子时钟，这些时钟由地面上的特定基站进行校准。卫星不断地广播其位置和时间戳，每个消息都带有它的本地时间。这种广播准许地面上的接收器精确地计算出它的位置。为了便于解释，我们首先假设所有(包括接收器的)时钟是同步的。

要计算位置，首先考虑二维的情况，如图 6.4 所示，其中画出了两个卫星，以及到每个卫星等距离的点圆。y 轴表示高度，而 x 轴表示与地球水平线平行的直线。忽略最高点，我们可以看到，两个圆的相交点是唯一的(在这种情况下，可能是在山上的某个地方)。

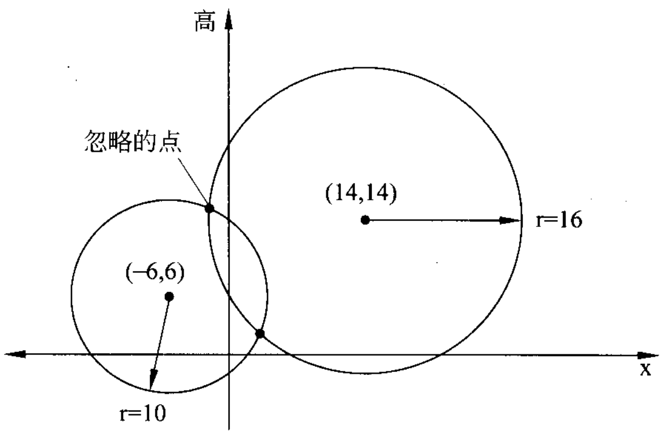


图 6.4 在二维空间中计算位置

相交圆原理可以扩展到三维空间，即我们需要三个卫星来确定地面上某个接收器的经度、纬度和海拔高度。这种定位方法都很简单，但如果假设时钟不再是同步的时候，问题就

变得复杂了。

有两个重要的现实情况我们需要考虑：

(1) 数据从卫星的位置到达接收器需要花费一定的时间。

(2) 接收器的时钟往往与卫星的时钟不同步。

假设卫星的时间戳是完全精确的。设 Δ_r 表示接收器与实际时间的偏差。如果来自卫星 i 的消息的时间戳为 T_i，那么接收器的 Δ_i 延时由两部分组成：实际延时和它自身的偏差：

$$\Delta_i = (T_{now} - T_i) + \Delta_r$$

由于信号是以光速 c 传播的，那么卫星的距离是 $c\Delta_i$。于是

$$d_i = c(T_{now} - T_i)$$

为接收器与卫星之间的时间距离，这个距离又可以写作 $d_i + c\Delta_r$。真正距离可以如下计算：

$$d_i = \sqrt{(x_i - x_r)^2 + (y_i - y_r)^2 + (z_i - z_r)^2}$$

其中 x_i、y_i 和 z_i 表示卫星 i 的坐标。于是，如果我们有 4 个卫星，就可以得到含 4 个未知数的 4 个方程，从而就可以求解出接收器的坐标 x_r、y_r 和 z_r 以及 Δ_r。换句话说，GPS 度量也可以给出实际时间。本章的后面我们还会回来讨论用类似方法确定位置。

现在，我们假定这种计算相当精确。当然，它们其实并不精确。因为，GPS 没有考虑闰秒。换句话说，这里有一个与 UTC 的系统偏差，即到 2006 年 1 月 1 日为 14 秒。这种差错在软件中很容易补偿。但是，还有很多其他差错来源，例如，卫星上的原子时钟并不总是同步的，卫星的位置并不是可以精确地知道，接收器的时钟的精确度也是有限的，信号传播的速度不是恒定的(例如，当信号进入电离层时速度会减慢)等等。而且，我们知道，地球不是很规则的球体，需要进一步的校正。

总的来说，要计算精确的位置并不是一件简单的事情，需要深入到很多细节中。然而，即使是相对便宜的 GPS 接收器，也可以定位在 1～5 米的范围内。而且，专业接收器()声称具有小于 20～35 纳秒的误差。有关的详细信息，可以参见(Zogg 2002)。

6.1.3 时钟同步算法

如果一台机器上有 WWV 接收器，那么我们需要使其他所有的机器与它同步。如果没有一台机器有 WWV 接收器，每台机器都按自己的机器时间计时，这时我们的目标是尽可能使所有机器的时间保持同步。专家们已经提出了许多同步算法(例如 Cristian 1989，Drummond 和 Babaoglu 1989，Kopetz 和 Ochsenreiter 1987)。在文献(Ramanathan 等 1990)中给出了各种算法的综述。

所有算法都有相同的系统基础模型，我们下面对其进行描述。假设每台机器都有一个每秒产生 H 次中断的计时器。当计时器产生中断时，中断处理程序将软件时钟加 1，软件时钟记录从过去某一约定时间开始的滴答(中断)数。我们将这个时钟值称为 C。更具体地说，当 UTC 时间为 t 时，机器 P 上的时钟值为 Cp(t)。最理想的情况是对所有的 p 和 t，都有 Cp(t)=t；换言之，dC/dt 的理想值为 1。

真正的计时器并不是每秒精确地中断 H 次。理论上，当 H=60 时，计时器应每小时产

生 216 000 个滴答。实际上，现代计时器芯片所能达到的错误率大约是 10－5，这意味着一台具体机器的计时器每小时可以产生的滴答次数在 215 998 到 216 002 之间。更准确地说，若存在某一常数 ρ，使得：

$$1-\rho\leqslant\frac{dC}{dt}\leqslant1+\rho$$

那么可以说计时器在它的规定范围内工作。这个常量 ρ 由厂商规定，称为**最大偏移率**(maximum drift rate)。稍慢的、精确的及稍快的时钟如图 6.5 所示。

若两个时钟以相反的方向偏离 UTC，在它们同步后的 Δt 时，它们间的最大可能差值是 $2\rho\Delta t$。如果操作系统设计者要保证每两个时钟之间的差值不超过 δ，那么时钟必须至少每 $\delta/2\rho$ 秒重新同步一次。各种算法的差异在于实现重新同步的方法不同。

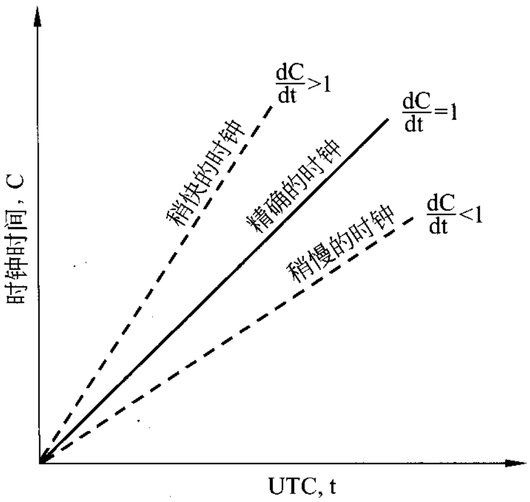


图 6.5　当时钟以不同的速率滴答时，时钟时间和 UTC 之间的关系

6.1.3.1　网络时间协议

很多协议中常用的方法是由 Cristian(1989)提出的，那就是让客户与**时间服务器**(time server)联系。时间服务器可以精确地提供当前时间，因为它装备了一个 WWV 接收器或一个精确的时钟。当然，问题是，何时与该服务器联系，消息延时也会使得报告的时间过时。其中的技巧是对这些延时做出很好的估计。请看图 6.6 所描绘的情况。

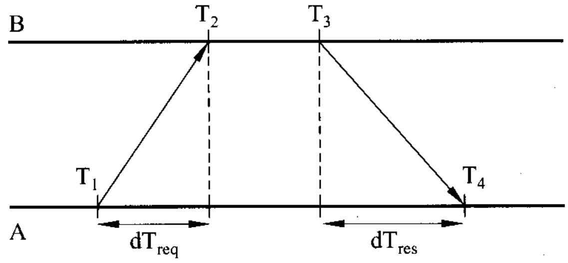


图 6.6　从一个时间服务器得到当前时间

在这种情况下，A 发送一个请求给 B，时间为 T_1。反过来，B 会记录接收时间 T_2(依据自己的时钟)，并返回一个响应消息，时间为 T_3，该时间比 T_2 晚。最后，A 记录下响应消息达到的时间 T_4。假设从 A 到 B 的传输延时与从 B 到 A 的大致相同，即 $T_2-T_1\approx T_4-T_3$。这样，A 就可以计算出与 B 的时间偏差：

$$\theta=T_3-\frac{(T_2-T_1)+(T_4-T_3)}{2}=\frac{(T_2-T_1)+(T_3-T_4)}{2}$$

当然，时间是不准许往后退的。如果 A 的时钟快了，$\theta<0$，表示原则上 A 应该把它的时钟往后调。这是不准许的，因为这将会导致严重的问题，例如，如果某个目标文件正好是在时钟调整后编译的，那么它的时间就比在时钟调整前修改的源文件要早了。

这种修改必须逐步进行。一种方法如下。假设把计时器设置为每秒产生 100 个中断。

正常情况下，每个中断将添加10毫秒。当减慢时，每个中断例程只添加9毫秒，直到校正完成。同样，通过在每个中断中添加11毫秒，时钟也可以逐步往前调。

网络时间协议(network time protocol, NTP)在服务器之间创建了两条连接。换句话说，B也可以探查A的当前时间。偏差θ的计算同上，而延时δ的计算如下：

$$\delta = \frac{(T_2 - T_1) + (T_4 - T_3)}{2}$$

可以先将每个(θ, δ)对缓存起来，最后以δ的最小值作为两个服务器之间的延时，从而，相应的θ值就是最可靠的偏差。

原则上，对称地应用NTP也可以让B参照A来调整它的时钟。但是，如果已知B的时钟更精确，那么这种调整就是愚蠢的了。要解决这个问题，NTP把服务器分成多个层。含有**参考时钟**(reference clock)的服务器(如WWV接收器或原子时钟)称为**1层服务器**(stratum-1 server)(时钟本身为0层)。当A与B联系时，如果它的层比B的要高，那么它只调整自己的时间。而且，经过同步化后，A将比B高一层。换句话说，如果B是k层服务器，且A的初始层已经大于k，那么，经过时间调整后，A就变成了(k+1)层服务器。由于NTP的对称性，如果A的层数比B的低，那么B将按照A来调整自己。

NTP有很多重要的特性，其中很多是与错误的识别和屏蔽有关，还有一些是与安全攻击有关。有关NTP的描述见(Mills 1992)，可以获得1～50毫秒内的精确度。最新版本的NTPv4最初只是介绍了它的实现方法，但在(Mills 2006)中进行了详细描述。

6.1.3.2 Berkeley算法

在很多算法(如NTP)中，时间服务器是被动的。其他机器定期向时间服务器询问时间，时间服务器所做的就是回答它们的请求。Berkeley UNIX系统采取了完全相反的方法(Gusella和Zatti 1989)。该系统中的时间服务器(实际上是时间守护程序)是主动的，它定期地询问每台机器的时间。基于这些回答，它计算出一个平均时间，并告诉所有其他机器将它们的时钟拨快到一个新的时间，或者拨慢，直到某个指定的减少量达到为止。这种方法适合于没有WWV接收器的系统。时间守护程序的时间必须由操作者定期手工设置。图6.7描述了这种方法。

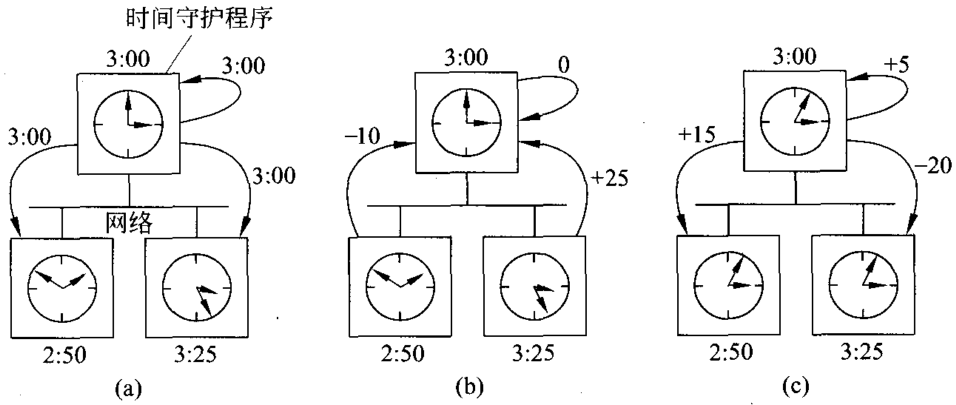


图6.7 Berkeley时钟同步算法

(a) 时间守护程序向所有其他机器询问时钟值；(b) 其他机器做出应答；

(c) 时间守护程序通知每台机器如何调整它们的时钟

在图 6.7(a)中，在 3：00 时，时间守护程序把它的时间告诉其他机器，并询问它们各自的时间。在图 6.7(b)中，各台机器将它们各自的时间与时间守护程序时间的差值告诉时间守护程序。有了这些值，时间守护程序计算出它们的平均值，并通知各台机器如何调整各自的时钟，如图 6.7(c)所示。

很多情况下，所有机器达成同一时间就足够了，没有必要使它们的时间与广播中所播的时间一致。在图 6.7 中的示例中，不会手工调整守护程序的时钟，假使其他结点都没有与外部计算机通信，也不会有什么危害。人人都乐意与当前时间达成一致，而不关心这个值是否与实际的时间有何联系。

6.1.3.3　无线网络中的时钟同步

更传统的分布式系统中一个重要的优点是，我们可以很容易而且很有效地部署时间服务器。而且，大多数机器可以相互联系，准许信息的相对分散。这些假设在很多无线网络(最著名的就是传感器网络)中就不再有效了。结点是受资源限制的，多跳路由很昂贵。而且，针对能量消耗的优化算法往往很重要。这些以及其他因素使得需要为无线网络设计不同时钟的同步化设计算法。下面我们将介绍一种解决方法。Sivrikaya 和 Yener(2004)给出了其他解决方法的简要介绍。在 Sundararaman 等(2005)中进行了扩展介绍。

参考广播同步化(Reference broadcast synchronization，RBS)是一种时钟同步化协议，它与其他协议很不相同(Elson 等 2002)。首先，该协议没有假设具有精确时间的结点。它不是为所有结点提供 UTC 时间，而只是在内部实现时钟的同步化，就像 Berkeley 算法一样。第二，到目前为止我们所讨论的方法是用来为发送器和接收器实现同步，基本上是遵循双向协议。RBS 与这种模式不同，它只是使接收器同步化，发送器则位于该循环之外。

在 RBS 中，发送器广播一个参考消息，让接收器调整它们的时钟。一个关键因素是，在传感器网络中，信号到其他结点的时间大致是固定的。在这种情况下的传播时间是从消息离开发送器的网络接口那一刻开始计算的。因此，消息传输中两个重要的变源(即构建消息所花费的时间和访问网络所花费的时间)对延时的计算不再其作用了。其原理如图 6.8 所示。

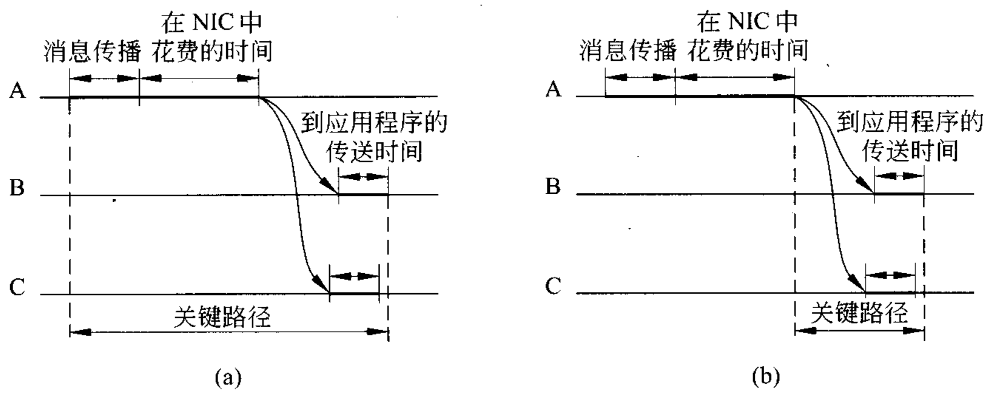


图　6.8

(a) 通常情况下确定网络延时的关键路径；(b) 在 RBS 情况下的关键路径

注意，在诸如 NTP 的协议中，消息在网络接口中传输是就把时间添加到其中了。而且，由于无线网络是基于内容的协议，通常无法确定消息在真正传输之前已经花费了多少时

间。这些不确定因素在 RBS 中都被去除了。所留下的是在接收器中的传送时间，但这个时间比网络访问时间小很多。

RBS 的基本思想很简单：当一个结点广播一个参考消息 m 时，每个结点 p 只是记录它接收到 m 的时间 $T_{P,M}$。注意，$T_{P,M}$是从 p 的本地时钟读取的。忽视时钟偏差，两个结点 p 和 q 就可以交换各自的传送时间，以便计算相互的相对偏差：

$$偏差[p,q] = \frac{\sum_{k=1}^{M}(T_{p,k} - T_{q,k})}{M}$$

其中，M 是发送的参考消息总数。这个信息很重要：结点 p 将知道 q 的时钟值相对于自己时钟值的偏差。而且，如果它存储了这些偏差，就没有必要调整自己的时钟，这就可以节省能量消耗。

但是，时钟是会变化的。这导致的结果是，上面计算的平均偏差将不再有用：最后发送的值没有第一个精确。而且，随着时间的流逝，偏移量将增加。Elson 等使用了一种非常简单的方法来进行补偿：他们不是计算平均值，而是应用标准线性回归来计算偏差，如下面函数所示：

$$偏差[p, q](t) = \alpha t + \beta$$

常数 α 和 β 可以从($T_{p,k}$，$T_{q,k}$)对计算出来。这种新方法可以通过结点 p 来更精确地计算出 q 的当前时钟，反之亦然。

6.2 逻辑时钟

至此，我们假设时钟同步化是与实际时间相关的。但是，在许多应用中，只要所有的机器具有相同的时间就够了，这个时间不一定要与实际时间相同。我们可以再进一步。例如，要运行 make 程序，两个结点可以这样对 input. o 过时与否达成一致：通过一个新版本的 input. c。在这种情况下，跟踪各自的事件（如生成一个新版本的 input. c）就是可以了。这类算法通常将时钟称为**逻辑时钟**（logical clock）。

在一篇著名的论文中，Lamport（1978）阐明，尽管时钟同步是可能的，但它不是绝对必要的。如果两个进程不进行交互，那么它们的时钟也无需同步，这是因为即使没有同步也觉察不出来，并且也不会产生问题。他进一步指出，通常，重要的不是所有的进程在时间上完全一致，而是它们在事件的发生顺序上要达成一致。在 make 例子中，有用的信息是 input. c 的创建时间比 input. o 的创建时间早还是晚，而不是它们创建的绝对时间。

本节我们将讨论 Lamport 算法，这是一个逻辑时钟同步化的算法。我们还要讨论一个称为向量时间戳的 Lamport 扩展算法，Lamport 在文献（Lamport 1990）中扩展讨论了这一算法。

6.2.1 Lamport 逻辑时钟

为了同步逻辑时钟，Lamport 定义了一个称为“**先发生**”（happens-before）的关系。表达

式 a→b 读作“a 在 b 之前发生”，意思是所有进程一致认为事件 a 先发生，然后事件 b 才发生。这种先发生关系有两种情况：

(1) 如果 a 和 b 是同一个进程中的两个事件，且 a 在 b 之前发生，则 a→b 为真。

(2) 如果 a 是一个进程发送消息的事件，而 b 为另一个进程接收这个消息的事件，则 a→b 也为真。消息不可能在发送之前就被接收，也不能在发送的同时被接收，这是因为消息需要一定时间才能到达接收端。

先发生关系是一个传递关系，所以若 a→b 且 b→c，则 a→c。如果事件 x 和 y 发生在两个互不交换消息的进程中（也不通过第三方间接交换消息），那么 x→y 不真，y→x 也同样不真。这两个事件称为**并发的**(concurrent)，这意味着无法说（或者不必说）这两个事件什么时候发生，哪个事件先发生。

我们需要一种测量时间的方法，使得对于每个事件 a，我们都能为它分配一个所有进程都认可的时间值 C(a)。这些时间值必须具有如下性质：如果 a→b，那么 C(a)<C(b)。重述我们前面陈述的情况，若 a 和 b 是同一进程中的两个事件，且 a 在 b 之前发生，则 C(a)<C(b)。与此类似，如果 a 是一个进程发送消息的事件，而 b 是另一个进程接收该消息的事件，那么 C(a)和 C(b)必须赋予大家都认可的具有 C(a)<C(b)关系的值。另外，时钟时间值 C 必须总是前进（增加），不能倒退（减少）。校正时间的操作是给时间加上一个正值，而不能是减掉一个正值。

现在我们来看一看 Lamport 算法怎样为事件分配时间。考虑如图 6.9(a)所示的三个进程。各个进程运行在不同的机器上，每台机器都有自己的时钟，它们以各自不同的速率工作。如图 6.9 所示，当进程 0 的时钟滴答了 6 次时，进程 1 的时钟滴答了 8 次，进程 2 的时钟则滴答了 10 次。每个时钟均以一个不变的速率工作，但是由于晶体之间的差异，各时钟的工作速率不同。

在时刻 6，进程 0 将消息 A 发送给进程 1，消息的传输时间取决于信任哪个时钟。不管怎样，当它到达时，进程 1 的时钟值为 16。如果消息携带有自身的起始时间 6，那么进程 1 将会推算该消息从进程 0 到进程 1 需要 10 个滴答。这个值的确是可能的。依此类推，消息 B 从进程 1 到进程 2 需要 16 个滴答，这也是可能的。

现在出现了有趣的问题。从进程 2 到进程 1 的消息 C 在时刻 60 离开，却在时刻 56 到达。与之类似，从进程 1 到进程 0 的消息在时刻 64 离开，却在时刻 54 到达。这些值显然是不可能的。必须防止这种情况出现。

Lamport 的解决方法直接遵循先发生关系。既然 m_3 在时刻 60 离开，那么它只能在时刻 61 或更晚的时刻到达。所以，每个消息都应携带依据发送者时钟的发送时间。当消息到达并且接收者时钟显示的时间值比消息的发送时间早时，接收者就将它的时钟调到一个比发送时间大 1 的值。在图 6.9(b)中，我们看到 m_3 现在到达的时间是 61。与此类似，m_4 到达的时间是 70。

为了准备有关向量时钟的讨论，让我们来更精确地规则化这个过程。这里，很重要的一点是，要区分在第 1 章中所介绍过的软件的三个不同层：网络层、中间件层和应用层，如图 6.10 所示。这是中间件层的一个典型部分。

要实现 Lamport 逻辑时钟，每个时钟 P_i 维护一个局部计数器 C_i。这些计数器按如下步骤进行更新(Raynal 和 Singhal 1996)：

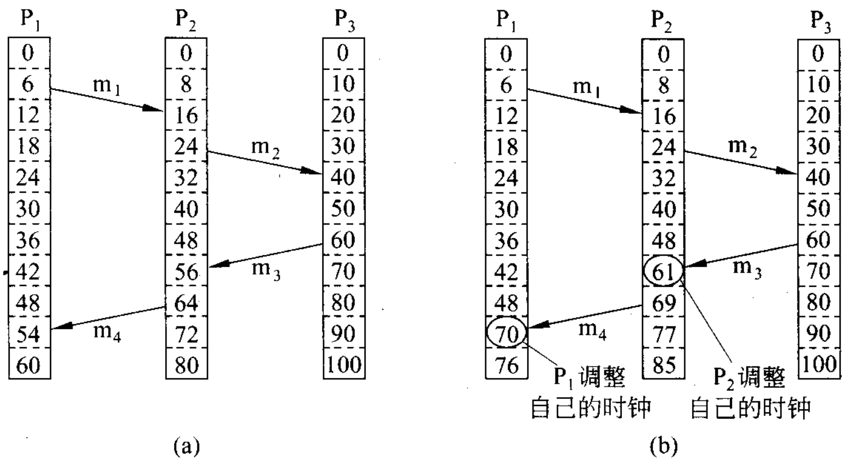


图 6.9 Lamport 算法校正三个进程的不同时钟

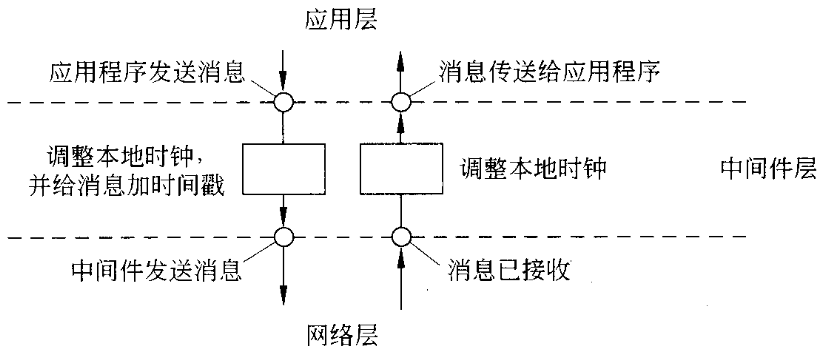


图 6.10 在分布式系统中 Lamport 逻辑时钟的定位

(1) 在执行一个事件之前（如在网络上发送一个消息，传送一个消息给应用程序，或者其他内部事件），P_i执行 $C_i \leftarrow C_i+1$。

(2) 当进程 P_i发送一个消息 m 给 P_j时，在执行前面的步骤后，把 m 的时间戳 ts(m)设置为等于 C_i。

(3) 在接收消息 m 时，进程 P_j调整自己的局部计数器为 $C_j \leftarrow \max\{C_j, ts(m)\}$。然后，执行第一步，并把该消息传送给应用程序。

在某些情况下，还需要一个附加条件：即两个事件不会完全同时发生。为了实现这个目标，我们可以将事件发生所在的进程号附加在时间的低位后，并用小数点分开。例如，进程 P_i中的事件都发生在时刻 40，那么就记为 40. i。

注意，如果 a 发生在进程 P_i的 $C_i(a)$时刻，那么，通过把事件时间赋值为 $C(a) \leftarrow C_i(a)$，就可以得到我们最初寻找的全局时间值的分布式实现。

示例：全序多播

作为 Lamport 时间戳的一个应用，让我们来考虑一下已经把一个数据库复制到几个站点的情况。例如，为了提高查询的效率，银行可能在两个不同的城市都保存一个账户数据库的副本，例如纽约和旧金山。一个查询总是被发送到最近的那个副本。查询的快速应答在一定程度上是以较高的更新开销为代价的，这是因为每次更新操作都必须在每个副本上执行。

事实上,对更新的要求更严格。假设一位旧金山的客户想在他的账户中存进100美元,且当前他账户上有1000美元。与此同时,纽约的银行职员要向该账户加进1%的利息。此数据库的两个副本都应该执行这两次更新操作。但是,由于网络中的传输延迟,这两次更新可能以图6.11所示的顺序到达。

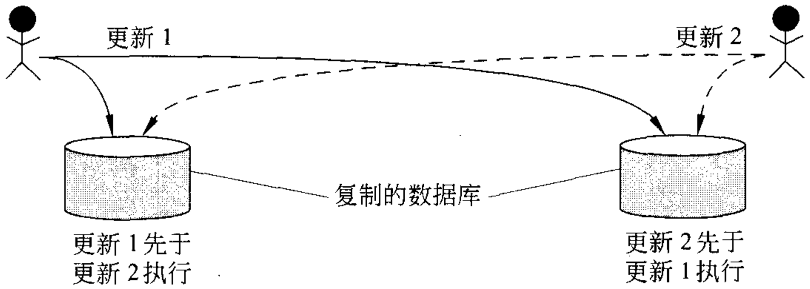


图6.11 更新一个复制的数据库使它处于了一种不一致的状态

在旧金山,客户存款数的更新操作是在利息加入之前完成的。相反,纽约的账户副本则是先加入了1%的利息,然后存入了100美元。结果,旧金山的数据库记录的存款总数是1111美元,而纽约的数据库记录的存款总数是1110美元。

我们面临的问题是在每个副本上的两个更新操作应该以相同的顺序执行。尽管先存款还是先更新利息有很大的不同,但是对于一致性来说,遵循哪个顺序都是无关紧要的。重要的是两个副本要完全相同。一般来说,像上面的情况需要进行一次**全序多播**(totally-ordered multicast),即一次将所有的消息以同样的顺序传送给每个接收的多播操作。Lamport时间戳可以用于以完全分布式的方式实现全序多播。

考虑一组彼此互相多播消息的进程。每个消息都以它的发送者的当前逻辑时间作为时间戳。当一个消息被多播时,理论上它也被传送给它的发送者。另外,我们假定来自同一个发送者的消息以它们被发送的顺序被接收,并且没有消息丢失。

进程接收到一个消息后,将它放进一个本地队列中,并根据它的时间戳进行排序。然后接收者向其他所有进程广播一个确认消息。如果我们按照Lamport算法来校正本地时钟,那么接收到的消息的时间戳总是早于确认消息的时间戳。该方法令人感兴趣的地方是所有进程最终将有相同的本地队列副本(就好像没有消息被删除一样)。

只有当队列中的一个消息处于队列头,并且已经被所有其他进程确认了时,进程才可以将它传送给运行中的应用程序。此时,把消息从队列中删除并传送给应用程序,相关的确认消息可以被简单地删除。因为每个进程都有相同的队列,所以在任何地方,所有的消息都以相同的顺序交付。换言之,我们已经建立了全序多播。

正如我们将在后面章节中要看到的那样,对复制服务来说,为了让这些服务在任何地方都可以以相同的顺序执行相同的操作,全序多播是一个重要的工具。由于复制所遵循的处理过程基本上与相同的有限状态机相同,因此又称之为**状态机复制**(state machine replication)(Schneider 1990)。

6.2.2 向量时钟

Lamport逻辑时钟导致分布式系统中的所有事件都要经过排序以具有这样的性质:如

果事件 a 在事件 b 之前发生，那么 a 也应该排在 b 之前，即 $C(a)<C(b)$。

然而，使用 Lamport 时间戳后，只通过比较事件 a 和 b 各自的时间值 $C(a)$ 和 $C(b)$，无法说明它们之间的关系。换句话说，$C(a)<C(b)$ 不能说明事件 a 就是在事件 b 之前发生。还需要另外一些信息。

为了便于介绍，我们来考虑如图 6-12 所示的三个进程发送的消息。$T_{snd}(m_i)$ 表示在该逻辑时间发送消息 m_i，同样，$T_{rcv}(m_i)$ 表示在该逻辑时间接收消息 m_i。这样，我们就知道，对每个消息有 $T_{snd}(m_i)<T_{rcv}(m_i)$。但从 $T_{snd}(m_i)<T_{rcv}(m_i)$ 中我们可以得出什么结论呢？

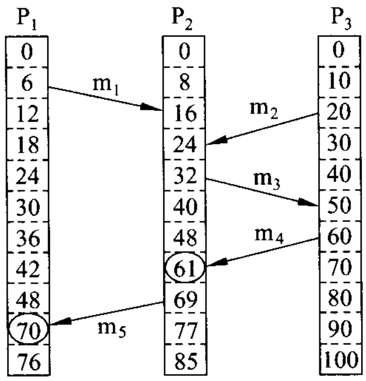


图 6-12　使用逻辑时钟进行并发消息传送

这里，如果 $m_i=m_1$ 且 $m_j=m_3$，那么我们就可以知道，这些值对应着进程 P_2 发生的事件，说明 m_3 的确是在接收消息 m_1 之后发送的。这表明，消息 m_3 的发送与消息 m_1 的接收有关。我们还知道 $T_{rcv}(m_1)<T_{snd}(m_2)$，但是，消息 m_2 的发送与消息 m_1 的接收无关。

问题在于 Lamport 时间戳不能捕获**因果关系**(causality)。因果关系可以通过**向量时钟**(vector clock)来捕获。分配给事件 a 的向量时钟 $VC(a)$ 具有下列性质：如果对某一事件 b，有 $VC(a)<VC(b)$，那么认为事件 a 在因果关系上先于事件 b。向量时钟的创建是通过让每个进程 P_i 维护一个向量 VC_i 来完成的，该向量具有下面两个性质：

(1) $VC_i[i]$ 是到目前为止进程 Pi 发生的事件的数量；

(2) 如果 $VC_i[j]=k$，那么进程 P_i 知道进程 P_j 中已经发生了 k 个事件。因此，P_i 知道 P_j 的逻辑时间。

第一个性质是通过在进程 P_i 中的新事件发生时递增 $VC_i[i]$ 来维护的。第二个性质是通过在所发送的消息中携带向量来维护的。特别地，执行以下步骤：

(1) 在执行一个事件之前（如在网络上发送一个消息，传送一个消息给应用程序，或者其他内部事件），P_i 执行 $VC_i[i] \leftarrow VC_i[i]+1$。

(2) 当进程 P_i 发送一个消息 m 给 P_j 时，在执行前面的步骤后，把 m 的时间戳 $ts(m)$ 设置为等于 VC_i。

(3) 在接收消息 m 时，进程 P_j 通过为每个 k 设置 $VC_j[k] \leftarrow \max\{VC_j[k], ts(m)[k]\}$ 来调整自己的向量。然后，执行第一步，并把该消息传送给应用程序。

注意，如果事件 a 具有时间戳 $ts(a)$，那么 $ts(a)[i]-1$ 表示在进程 P_i 中先于 a 所处理的事件数量。因此，当进程 P_j 从进程 P_i 接收一个时间戳为 $ts(m)$ 的消息时，它知道进程 P_i 在发送消息 m 之前已经发生了多少事件。更为重要的是，进程 P_j 还知道在进程 P_i 发送消息 m 之前其他进程发生了多少事件。换句话说，时间戳 $ts(m)$ 告诉接收者其他进程中有多少事件发生在发送消息 m 之前，并且 m 可能在因果关系上与哪些事件有关。

6.2.2.1　强制因果有序通信

使用向量时钟，就有可能确保所有先于某个消息的所有消息接收后才传送这个消息。要使这种方案可行，我们假设消息是在一组进程中进行多播的。注意，**因果有序多播**

(causally-ordered multicasting)比我们前面讨论的全序多播更弱。尤其是如果两个消息互相没有任何关系时,并不用关心以哪种顺序传送给应用程序。它们可能在不同的地方以不同的顺序进行传送。

而且,我们还假设只在发送和接收消息时才调整时钟。特别是在发送消息时,进程 P_i 只将 $VC_i[i]$增加 1。当它接收一个时间戳为 ts(m)的消息 m 时,它只对每个 k 将 $VC_j[k]$在 max{$VC_j[k]$, ts(m)[k]}之间进行调整。

现在,假设进程 P_j从进程 P_i接收一个(向量)时间戳为 ts(m)的消息 m。把该消息传送到应用层将被延时,直到以下两个条件满足为止:

(1) $ts(m)[i] = VC_j[k]+1$

(2) $ts(m)[k] \leqslant VC_j[k]$,其中 $k \neq i$

第一个条件表示,消息 m 是进程 P_j希望从进程 P_i接收的下一条消息。第二个条件表示,当进程 P_i接收发送消息 m 时,P_j已经接收到了所有来自进程 P_i的消息。注意,这里没有必要让进程 P_j延迟其自己消息的传送。

作为一个示例,来看看图 6.13 所示的三个进程 P_0、P_1和 P_2。在本地时间(1, 0, 0)进程 P_0发送消息 m 到其他两个进程。该消息由进程 P_1接收后,进程 P_1决定发送消息 m*,它比 m 先到达进程 P_2。此时,m* 的传送被进程 P_2延迟,直到 m 被接收且传送给了 P_2的应用层。

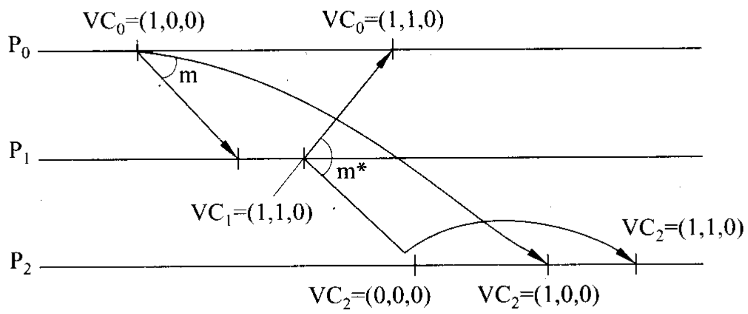


图 6.13 强制因果通信

6.2.2.2 关于有序消息交付的说明

一些中间件系统,尤其是 ISIS 和它的后继 Horus(Birman 和 van Renesse 1994),提供了对全序和因果排序的可靠多播的支持。对于应由消息通信层支持排序还是由应用层处理排序(请参见 Cheriton 和 Skeen 1993;Birman 1994)存在一些争议。事情还没有解决,更重要的是,这种争议今天仍存在。

让通信层处理消息排序存在两个主要问题。首先,因为通信层不能判断消息包含的内容,所以只能捕获潜在的因果关系。例如,来源相同的两个完全独立的消息总是被通信层标记为因果相关的。这种方法过于严格,可能会导致效率问题。

第二个问题是,并不是所有的因果关系都可以捕获。假设 Alice 张贴了一篇文章。如果随后她打电话给 Bob,告之她刚才所写的内容,Bob 可能没有看见 Alice 发在新闻系统上的文章就发了一篇文章作为回复。换句话说,由于外部通信,Bob 和 Alice 的张贴之间有了因果关系。这个因果关系不会被网络新闻系统所捕获。

本质上,像许多其他特定于应用的通信问题一样,排序问题可以根据通信的具体应用而得到充分的解决。在系统设计中,这也称为**端对端参数**(end-to-end argument)(Saltzer 等 1984)。只具有应用层解决方案的系统的一个缺点是,开发者不得不关注与应用系统的核心功能不直接相关的问题。例如,当开发一个诸如网络新闻这样的消息系统时,排序问题可能并不是最重要的问题。在这种情况下,让底层的通信层处理排序问题可能比较方便。我们还将多次提到端对端参数,尤其是在讨论分布式系统中的安全问题时。

6.3 互斥

分布式系统的基础是多进程之间的并发和协作。在很大情况下,这也就意味着进程将需要同时访问相同的资源。为了保证这种并发访问不会崩溃资源或使得它不一致,需要一些解决办法来保证进程的互斥访问。本节将介绍一些已发表的更为重要的分布式算法。有关互斥的分布式算法的论文见(Saxena 和 Rai 2003)。更旧些但仍参考价值的论文见(Velazquez 1993)。

6.3.1 概述

分布式互斥算法可以分为两种不同的类型。在**基于令牌的解决方法**(token-based solution)中,互斥是通过在进程之间传递一个特殊的消息来实现的,这个消息称为**令牌**(token)。这里只有一个令牌,拥有该令牌的进程就准许访问共享资源。完成后把该令牌传递给下一个进程。如果某个进程拥有该令牌但不需要访问资源,那么它就只是把该令牌往下传递。

基于令牌的解决方法有一些重要的特性。首先,按照进程的组织结构,可以很容易地确保每个进程都有机会访问资源。换句话说,它们避免了**饥荒**(starvation)。第二,由于简单,可以很容易地避免**死锁**(deadlock)。然而,基于令牌的解决方法的重大缺陷是很严重的一个:如果令牌丢失(例如,拥有令牌的进程崩溃了),就需要重新开启一个复杂的分布式进程,以确保创建一个新的令牌,但总之,这也是唯一的一个令牌。

另外,很多分布式遵循的是**基于许可的方法**(permission-based approach)。在这种情况下,一个进程如果需要访问资源,首先需要获得其他进程的许可。授予这样一种许可的方法有很多,下面章节将介绍其中一些。

6.3.2 集中式算法

在分布式系统中达到互斥的最直接的方法是仿照单处理器系统中的方法,选举一个进程作为协作者(例如,运行在具有最大网络地址号的机器上的进程)。无论何时一个进程要访问共享资源,它都要向协作者发送一个请求消息,说明它想要访问哪个资源并请求准许。如果当前没有其他进程访问资源,协作者就发送准许的应答消息,如图 6.14(a)所示。收到应答后,该请求进程即可访问资源。

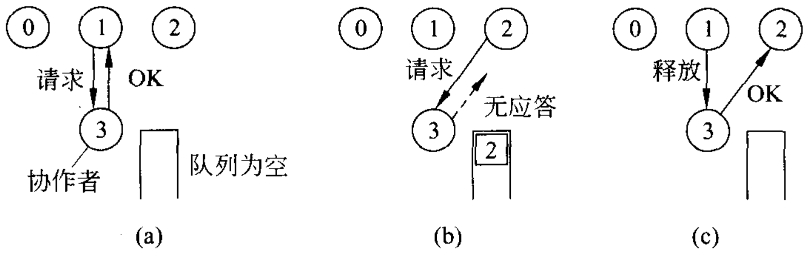


图　6.14

(a) 进程 1 请求协作者准许访问共享资源。请求得到了准许；(b) 然后进程 2 也请求准许共享资源。协作者不准许；(c) 进程 1 在释放资源时通知协作者，协作者然后对进程 2 做出应答

现在假设另一个进程，即进程 2，请求准许访问资源，如图 6.14(b)所示。协作者知道一个与此不同的进程正在访问资源，所以它不能准许该请求。拒绝进入的方法依系统而定。在图 6.14(b)中，协作者只是不进行应答，这样就阻塞进程 2，因为进程 2 正在等待应答。另一方面，协作者也可以发送一个“拒绝请求”的应答。不管采用哪种方法，协作者都将进程 2 的请求放到队列中，并等待更多的消息到来。

当进程 1 释放资源时，它向协作者发送一个消息以让协作者释放它的独占访问，如图 6.14(c)所示。该协作者从处于等待状态的请求队列中取出第一个进程，并向该进程发送一个允许进入消息。如果该进程仍然处于阻塞状态(也就是说这是发送给它的第一个消息)，它将解除阻塞并访问资源。如果已经明确发送一个消息拒绝准许，那么该进程将不断查询进入的消息，或者稍后进入阻塞状态。不管使用了哪种方法，当进程获得准许，它就能够访问资源。

显然，该算法保证了互斥的实现，即协作者在某一时刻只让一个进程访问资源。它也很公平，因为准许请求的顺序同接收它们的顺序是一致的。没有进程会处于永远等待状态(不会出现饿死的情况)。此方法也易于实现，对每个共享资源只需 3 条消息(请求、准许和释放)。对很大实际的解决方法，其简单性很有吸引力。

但是，集中式方法也存在缺点。协作者是一个单个故障点，所以如果它崩溃了，整个系统可能瘫痪。在一般情况下，如果进程在发出请求之后被阻塞，那么请求者就不能区分“拒绝进入”和协作者已经崩溃这两种情况，因为上述两种情况都没有消息返回。此外，在规模较大的系统中，单个协作者会成为性能的瓶颈。然而，其简单性带来的好处大于潜在的缺点。而且，正如后面示例所示的那样，分布式解决方法并不比这更好。

6.3.3　非集中式算法

只有单个的协作者往往是一种较差的方法。我们来看看一种非集中式的解决方法。(Lin 等 2004)提出了一种投票算法，它可以使用基于 DHT 的系统来运行。本质上，其解决方法是按如下方式扩展了集中式协作者。假定每种复制了 n 次。通过并发进程，每个副本有其自身的协作者用于控制访问。

但是，只要某个进程要访问资源，它只需从 $m>n/2$ 个协作者中获得多数投票即可。与前面介绍的集中式解决方法不同，我们假设，当某个协作者没有获得访问某个资源的许可时(如果访问许可授给了另外一个进程，就会这样)，它将告诉请求者。

这种算法主要是使得原来的集中式算法少受单个协作者失效的影响。它假设当某个协作者崩溃时，能快速地恢复，但会忘记崩溃前的所有投票。其风险是，重置将使得协作者忘记它前面已授权某进程访问资源的许可。因此，在其恢复后，可能错误地把该许可又赋给了另外一个进程。

设 p 为时间段 Δt 里协作者重置的概率，那么，在相同的时间段里，m 个协作者中有 k 个进行了重置的概率为：

$$p[k] = \begin{bmatrix} m \\ k \end{bmatrix} p^k (1-p)^{(m-k)}$$

假设要破坏投票算法的正确性，至少应有(2m－n)个协作者需要重置，那么发生这种情况的概率为 $\sum_{k=2m-n}^{n} P(k)$。为了加深读者对其含义的理解，假设我们正使用一个基于 DHT 的系统，其中每个结点分享 3 个小时时间。设 Δt 为 10 秒，这被认为是单个进程要访问共享资源的保守值。当 n＝32 且 m＝0.75n 时，违反正确性的概率小于 10^{-40}。该概率肯定小于所有资源的可用性。

为了实现这种算法，(Lin 等 2004)使用了一个基于 DHT 系统，其中一个资源被复制了 n 次。假设该资源具有唯一的名称 rname。那么就可以假设把第 i 个副本命名为 rname-i，使用一个已知的散列函数，就可以用 rname-i 来计算出一个唯一键值。因此，给定一个资源名，每个进程可以生成 n 个键值，从而可以查找出每个副本对应的结点（并控制对给副本的访问）。

如果对资源的访问被拒绝（即某个进程获得了少于 m 的投票），那么它将在一个随机选择的时间后再进行下一次尝试。这种方案的问题是，如果很多结点要访问同一资源，将证明其实用性大大降低。换句话说，如果太多的结点去竞争获得许可，最终使得谁也得不到足够的投票，从而使得资源谁也用不了。这个问题的解决方法可以在(Lin 等 2004)找到。

6.3.4 分布式算法

拥有单个故障点往往是不可接受的，所以研究者开发了分布式互斥算法。Lamport 在 1978 年发表的关于时钟同步的论文中首次提出了一种分布式互斥算法。Ricart 和 Agrawala(1981)对它作了进一步的改进。本节将讨论他们的方法。

Ricart 和 Agrawala 算法要求系统中的所有事件都是完全排序的。也就是说，对于每对事件，比方说消息，哪个事件先发生必须非常明确。6.2.1 节中给出的 Lamport 算法是一种完成这种排序的方法，该算法还能用于为分布式互斥提供时间戳。

该算法的工作过程如下：当一个进程要访问一个共享资源时，它构造一个消息，其中包含它要访问的资源名、它的进程号和当前（逻辑）时间。然后，它将该消息发送给所有其他的进程，理论上讲也包括它自己。假设消息的传送是可靠的，也就是说，没有消息丢失。

当一个进程接收到来自另一个进程的请求消息时，它根据自己与消息中的资源相关的状态来决定它要采取的动作。可以分为三种情况：

(1) 若接收者没有访问资源，而且也不想访问它，就向发送者发送一个 OK 消息。

(2) 若接收者已获得对资源的访问，那么它就不进行应答，而是将该请求放入队列中。

(3) 如果接收者想访问资源但尚未访问时，它将收到的消息的时间戳与包含在它发送给其他进程的消息中的时间戳进行比较。时间戳最早的那个进程获胜。如果收到的消息的时间戳比较早，那么接收者向发送者发回一个 OK 消息。如果它自己的消息的时间戳比较早，那么接收者将收到的请求放入队列中，并且不发送任何消息。

发送许可请求后，进程进行等待，直到其他所有进程都给予了许可消息为止。一旦得到所有进程的许可，它就可以继续了。当它完成后时，它向其队列中的所有进程发送 OK 消息，并从队列中删除。

我们来理解一下该算法的工作原理。如果没有冲突，显然，该算法正常工作。然而，假设两个进程同时试图访问同一资源，如图 6.15(a)所示。

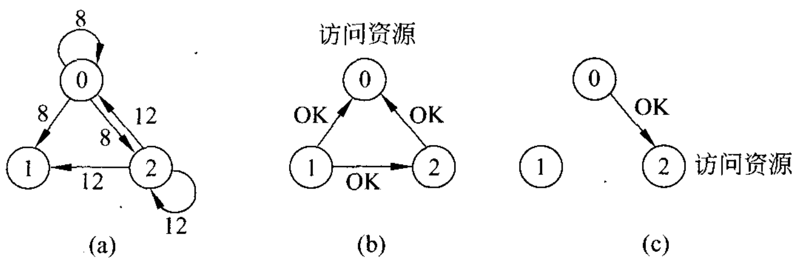


图 6.15

(a) 两个进程同时要访问一个共享资源；(b) 进程 0 具有最早的时间戳，所以它获胜；
(c) 当进程 0 完成时，它发送一个 OK 消息，所以进程 2 继续

进程 0 给每个进程发送一个时间戳为 8 的请求，然而，与此同时，进程 2 给每个进程发送了一个具有时间戳 12 的请求。进程 1 不想访问该资源，所以它给两个发送者都发回 OK 消息。进程 0 和进程 2 都发现了它们之间有冲突，都比较时间戳。进程 2 发现它失败了，所以它向进程 0 发送 OK 消息准许其访问。进程 0 现在把来自进程 2 的请求放入它的队列中，以便以后处理，然后访问该资源，如图 6.15(b)所示。当进程 0 完成时，它将进程 2 的请求从队列中删除，并给进程 2 发送 OK 消息，准许进程 2 继续，如图 6.15(c)所示。该算法之所以能工作，是因为它规定了在产生冲突的情况下，具有最小时间戳的进程获胜，并且每个进程都对时间戳的顺序达成了一致。

注意，在图 6.15 中，如果进程 2 比进程 0 早发送消息，使得进程 0 在自己发送请求前收到了进程 2 的请求，并准许进程 2 访问资源，那么有根本的不同。在这种情况下，进程 2 在接收到进程 0 的请求时，会发现自己已经获得对资源的访问，那么它就不会给进程 0 发送应答，而是将该请求放入自己的队列中。

与前面讨论的集中式算法一样，该算法实现的互斥也不会发生死锁或饿死现象。每次需要 2(n－1)个消息，这里的 n 是系统中的进程数目。这种算法的一个最大的优点是不存在单个故障点。

不幸的是，单个故障点被 n 个故障点所取代。如果任何一个进程崩溃，它就不能回答请求。这种不应答被错误地解释为拒绝请求，这样就阻塞了所有请求进入所有临界区的后续进程。因为 n 个进程之一发生故障的可能性至少是单一协作者发生故障的 n 倍，所以这样算法比前面的要糟糕 n 倍，同时要求有更多的网络流量。

该算法的这个弱点可用我们前面提到的技巧来修补。当请求到达时，接收者无论是准许还是拒绝都发送应答。一旦请求或应答消息丢失，发送者超时，它就继续发送请求消息，

直到收到一个应答消息为止，或者直到发送者得出目的进程已崩溃的结论为止。在一个请求被拒绝后，发送者将进入阻塞状态，以等待OK消息。

该算法的另一个问题是，要么必须使用多播通信原语，要么每个进程都必须自己维护组成员的列表，该列表中包括进入组的进程、离开组的进程以及崩溃的进程。该方法最适用于进程数目较少并且从不改变组成员的情况。

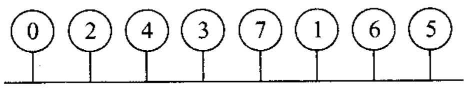
最后，回顾集中式算法存在的一个问题，即处理所有请求时会产生瓶颈问题。在分布式算法中，所有进程要参与决定共享资源的访问许可。即使有一个进程不能处理这种负载，那么让每个进程都并行地做完全相同的事情也是不可能的。

可以对该算法进行各种小的改进。例如，从所有进程获得访问许可其实并不必要。真正需要的是一种防止两个进程同时访问资源的方法。可以把该算法修改为，当一个进程从大多数进程获得许可（而不需获得所有进程的许可）时，它就可以获得访问许可。当然，在该算法的这种变体中，一个进程许可另一个进程后，它在第一个进程释放许可之前，是不能授给其他进程相同的许可了。

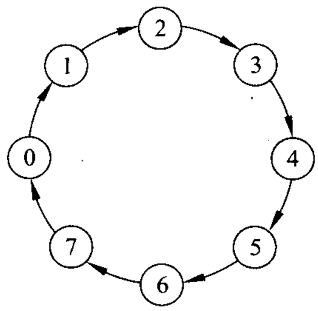
但是，这种算法与原来的集中式算法相比更慢、更复杂、花费更高，而且更不健壮。既然如此，为什么还要研究该算法呢？原因之一是它说明了分布式算法至少也是可能实现的，这一点我们开始并不知道。另外，通过指出它的不足，可以鼓励未来的理论家们试着创造出更具实用性的算法。最后，就像吃菠菜和在高中学习拉丁语一样，可能会有某些东西对你是有好处的。这需要花些时间去发现。

6.3.5 令牌环算法

图6.16示意了在分布式系统中实现互斥的一种完全不同的方法。这里是一个总线式网络（例如以太网），如图6.16(a)所示，进程没有固有的顺序。在软件中，可以构造出一个逻辑环，在该环中，为每个进程分配了一个它在环中的位置，如图6.16(b)所示。环的位置可以按照网络地址或以其他方式来分配。按照什么顺序进行排列并不重要。重要的是每个进程要知道谁在它的下一个位置上。



(a)



(b)

图 6.16

(a) 网络中一组无序的进程；(b) 以软件方式构造的一个逻辑环

当环初始化时，进程0得到一个**令牌**（token）。该令牌绕着环运行，用点对点发送消息的方式把该令牌从进程k传到进程k+1（以环大小为模）。进程从它邻近的进程得到令牌后，检查自己是否需要访问资源。如果要，那么它就继续，完成它要做的工作，然后释放资

源。在该进程完成后，它沿着环继续传递令牌。这里不准许进程在释放资源后，使用同一令牌立即再次访问资源。

如果一个进程得到了邻近进程传来的令牌，但是它并不想访问资源，那么它就把令牌沿环往下传递。因而，当没有进程要访问资源时，令牌就绕环高速地传递。

该算法的正确性是显而易见的。在任何时刻都只有一个进程拥有令牌，所以实际上只有一个进程才能访问资源。由于令牌以预定顺序在进程间循环传递，所以不会发生饿死现象。一旦某个进程要访问资源，那么最差的情况是等到其他所有进程都使用了这个资源后。

同样，该算法也存在问题。如果令牌丢失了，那么它必须重新生成令牌。实际上，检测令牌丢失是很困难的，这是因为网络中令牌两次出现的时间间隔是不确定的。一小时没有发现令牌并不意味着它就丢失了，也许某个进程还在使用它。

如果有进程崩溃，该算法也会出现麻烦，但是恢复起来却比其他算法更容易。如果我们要求每个进程在收到令牌后发出确认信息，那么当一个进程的邻近进程试图把令牌传递给它但是没有成功时，就会检测到这个崩溃的进程。这时，就可以将这个崩溃的进程从组中删除，令牌的持有者将令牌沿环传给这个崩溃进程的下一个进程，如果必要，传给再下一个。当然，这样做就需要每个进程都维护当前环的配置信息。

6.3.6 四种算法的比较

下面简略地比较一下我们已讨论的四种互斥算法。在图 6.17 中列举了这四个算法及它们的三个关键性质：进程访问和释放资源所需的消息数目，访问资源的延迟(假设消息在网络中顺序传递)，以及每个算法存在的问题。

算法	每次进/出需要的消息数	进入前的延迟（按消息数）	问题
集中式	3	2	协作者崩溃
非集中式	3mk, k=1,2, ...	2m	会发生饥饿，效率低
分布式	2(n−1)	2(n−1)	任何进程崩溃
令牌环	1~∞	0~n−1	令牌丢失，进程崩溃

图 6.17 四种互斥算法的比较

集中式算法是最简单也最有效的。使用这种算法，进入和离开临界区一次只需要三个消息：一个请求进入消息、一个许可进入消息和一个释放退出消息。在非集中式算法中，这些消息需要 m 个协作者执行一次，而现在则可能需要进行多次尝试(为此引入了变量 k)。分布式算法要求(n－1)个请求消息，每个消息对应于其他的进程，以及(n－1)个授权消息，这样总共有 2(n－1)个。(我们假定只使用点对点的通信通道。)对令牌环算法，这个数是变化的。如果每个进程都总想进入临界区，那么令牌每传递一步就会导致一次进出，这样平均每进入一次临界区就需要一个消息。在另一种极端的情况下，有时令牌可能在环中绕行了几个小时也没有进程要进入临界区。在这种情况下，每进入一次临界区需要的消息数目是不确定的。

对于三种算法来说，从进程想进入一个临界区到它真正进入临界区的这段延迟时间也是不同的。当使用资源的时间很短时，产生延迟的主要因素是如何访问资源的实际机制。

当使用资源时间较长时，产生延迟的主要因素是等待其他进程使用的时间。在图6.17中显示的是前一种情况。在集中式算法中，进入一次临界区只需两个消息时间，而在非集中式算法中，则需要3mk个消息时间，其中k是需要进行的尝试次数。假设消息是一个接一个地发送，那么在分布式算法中需要2(n－1)个消息时间。对于令牌环算法，所需的时间从0(恰好接到令牌)到n－1(恰好刚刚释放令牌)个消息时间里变化。

最后，除非集中式算法外，其余三种算法在进程崩溃的情况下都损失惨重。为了避免进程崩溃造成整个系统瘫痪，必须引入一些专门的措施和额外的复杂性。很有讽刺意味的是，分布式算法甚至比集中式算法对进程的崩溃更敏感。在容错系统中，这些算法都不适用，但是如果进程不经常崩溃，这些算法还是可以使用的。非集中式算法对进程的崩溃不那么敏感，但进程可能会发生饥饿，需要特殊方法来保证其效率。

6.4 结点的全局定位

在分布式系统中，随着结点数目的增长，任意结点要与其他结点保持联系也就变得很困难。这对运行诸如路由、多播、数据放置、查找等之类的分布式算法很重要。我们已经见过不同的示例，其中，大型的结点集被组织成特定的拓扑结构，从而有助于这些算法的有效运行。本节将来看看另一种与计时问题有关的组织形式。

在**几何覆盖网络**(geometric overlay network)中，每个结点给予一个m维几何空间的位置，这样，在该空间中的两个结点之间的距离反映了实际的性能度量。最简单也最实用的示例是，结点间的距离对应于它们之间的延时。换句话说，给定两个结点P和Q，那么距离d(P, Q)反映的是消息从P到Q传输所花费的时间，反之亦然。

几何覆盖网络的应用很多。请看这样一种情况，服务器O中的Web站点复制给了因特网上的多个服务器S_1，……，S_k。当客户C从O请求一个网页时，O将把该请求转发给离C最近的服务器，也就是那个能做出最佳响应时间的那个服务器。如果C和每个副本服务器的几何位置都是已知的，那么O就选择d(C, S_i)最小的服务器S_i。注意，这种选择只需要在O做本地处理即可。也就是说，无需提取C与每个副本服务器之间的所有延时。

另一个示例是最佳副本替换，我们将在下一章详细讨论。再来看一个Web站点，它已收集了其客户的位置。如果该站点把其内容复制给K个服务器，那么它就可以计算出K个替换副本的最佳位置，使得客户到副本的平均响应时间最小。如果客户和服务器具有能反映结点间延时的几何位置，完成这种计算是不值得一提的。

最后一个示例来看看基于位置的路由(position-based routing)(Araujo和Rodrigues 2005;Stojmenovic 2002)。在这种方案中，只是使用定位信息来把消息转发给目的地。例如，一个很简单的路由算法是让每个结点把消息转发给离目的地最近的邻结点。可以很容易证明，这种算法无需汇集，只需用本地信息来做决定。这里无需传播链接信息给网络中的所有结点，就像常用的路由算法那样。

理论上，在m维几何空间定位一个结点需要m＋1个到位置已知的结点的距离度量。通过m＝2的情况就很容易明白这一点，如图6.18所示。假设结点P要计算自己的位置，它需要计算出到另外三个位置已知的结点的距离。

就像在 GPS 中一样，通过求解如下三个等式(其中含有两个未知的 x_P 和 y_P)，结点 P 可以计算出自己的坐标(x_P, y_P)：

$$d_i = \sqrt{(x_i - x_p)^2 + (y_i - y_P)^2} \quad (i = 1, 2, 3)$$

d_i对应于 P 到位于(x_i, y_i)的结点的延时。该延时可以取往返延时延时的一半来估算，但应该明白的是，该值随时间的过去将会有所不同。P 每次重新计算其位置时，其结果都是不同的。而且，如果其他结点使用 P 的当前位置来计算它们的坐标，那么就应该清楚，P 位置的误差会影响其他结点定位的精度。

而且，还要明白的是，利用不同结点计算出的距离往往也是不一致的。例如，假设要计算图 6.19 所示的一维空间距离。在这个示例中，我们将看到，尽管 R 到 Q 的距离是 2.0，d(P, Q)为 1.0，但 d(P, R)却是 3.2，显然这与另两个测量不一致。

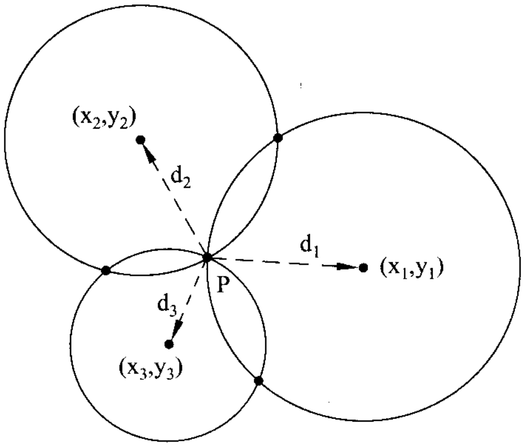


图 6.18 在二维空间中计算结点的位置

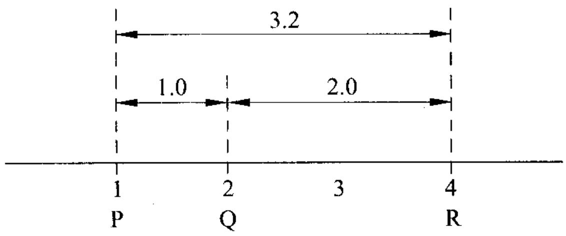


图 6.19 在一维空间中距离测量不一致

图 6.19 还提示了如何改进这种情况。在我们的示例中，只要在二维空间中计算其位置就可以解决这种不一致性。但是，如果要处理多个度量时，这往往不是一种通用的解决方法。事实上，考虑到因特网延时度量可能会违反**三角不等式**(triangle inequality)，通常不可能完全解决这种不一致性。三角表达式表示，在几何空间中，对任意的三个结点 P、Q 和 R，总有不等式 $d(P, R) \leq d(P, Q) + d(Q, R)$成立。

有多种方法来解决这些问题。(Ng 和 Zhang 2002)提出的一种方法是使用 L 个特殊结点 b_1，……，b_L，称为**地标**(landmark)。先度量每两个地标之间的延时 $d(d_i, b_j)$，然后用中心结点来计算每个地标的坐标。最后，中心结点寻求最小化下面的合计误差函数：

$$\sum_{i=1}^{L}\sum_{j=i+1}^{L}\left[\frac{d(b_i, b_j) - \hat{d}(b_i, b_j)}{d(b_i, b_j)}\right]^2$$

其中，$\hat{d}(d_i, b_j)$对应于几何距离，即结点 d_i和 b_j定位后的距离。

最小化合计误差函数中隐含的参数是维数 m。这里显然有 L>m，但我们可以为 m 选择一个比 L 小很多的值。在这种情况下，结点 P 度量它到 L 个地标中的每个地标的距离，并通过最小化如下算式来计算其坐标：

$$\sum_{i=1}^{L}\left[\frac{d(b_i, P) - \hat{d}(b_i, P)}{d(b_i, P)}\right]^2$$

事实证明，选择好地标，m 可以小至 6 或 7，而对任意结点 P 和 Q，$\hat{d}(P, Q)$与实际延时 $d(P, Q)$的差异不会大于因子系数 2(Szyamniak 等 2004)。

处理这种问题的另一种方法是把结点集视作为一个巨型系统，结点与其余每个结点之间是通过弹簧连接在一起的。在这种情况下，$|d(P, Q)-\hat{d}(P, Q)|$表明了结点 P 和 Q 被替换的程度与弹簧系统在静止状态有关。可以证明，通过让每个结点(轻微地)改变其位置，系统将趋向一个最佳组织结构，其中的合计误差是最小的。在 Vivaldi 中遵循的就是这种方法，详细内容请参见(Dabek 等 2004a)。

6.5 选举算法

很多分布式算法需要有一个进程作为协作者、发起者或者其他某种特殊的角色。通常，由哪个进程充当这个特殊的角色并不重要，重要的是它们中要有一个进程来充当。本节讨论用于选举出一个协作者的算法(这里使用协作者来作为那个特殊进程的通用名字)。

如果所有的进程都完全相同，没有任何区别，那么就无法选择其中的一个作为特殊进程。因此，我们假设每个进程有一个唯一的编号，例如它的网络地址(为了简单起见，我们假设每台机器只有一个进程)。通常，选举算法试着找出进程号最大的进程，将它指定为协作者。各个算法在找出这个进程时使用的方法有所不同。

此外，我们还假设每个进程都知道所有其他进程的进程号。但进程并不知道当前哪些进程正在运行，以及哪些进程崩溃了。选举算法的目标是确保发起选举后，当所有的进程都同意选出的新协作者时选举结束。这里有很多算法及其变体，在(Lynch 1996)和(Tel 2000)中介绍了几种重要的算法。

6.5.1 传统的选举算法

我们先来看看两种传统的选举算法，从而了解一下在过去的十来年里算法研究人员都在做些什么。在随后的章节中，我们将关注一下选举问题的新应用。

6.5.1.1 欺负算法

第一个例子来看看由 Garcia-Molina(1982)提出的**欺负算法**(bully algorithm)。当任何一个进程发现协作者不再响应请求时，它就发起一次选举。进程 P 按如下过程主持一次选举：

(1) P 向所有编号比它大的进程发送一个 ELECTION 消息；

(2) 如果无人响应，P 获胜并成为协作者；

(3) 如果有编号比它大的进程响应，则由响应者接管选举工作。P 的工作完成。

任何时候，一个进程只能从编号比它小的进程得到一个 ELECTION 消息。当有 ELECTION 消息到达时，接收者回送一个 OK 消息给发送者，表明它仍然在运行，并且接管选举工作。然后接收者主持一个选举，除非它正有一个选举。最终，除了一个进程外，其他

所有进程都将放弃，那个进程就是新的协作者。它将选举获胜的消息发送给所有进程，通知这些进程自己是新的协作者。

当一个以前崩溃了的进程现在恢复过来时，它将主持一次选举。如果该进程恰好是当前正在运行的进程中进程号最大的进程，它将赢得此次选举，接管协作者的工作。这样，最大的进程总是取胜，故称为"欺负算法"。

在图 6.20 中，我们看到了欺负算法如何工作的例子。该图显示了由 8 个进程组成的进程组，编号从 0 到 7。以前进程 7 是协作者，但是它崩溃了。进程 4 第一个注意到这一点，所以它发送 ELECTION 消息给所有比它大的进程，即进程 5、6、7，如图 6.20(a)所示。进程 5 和 6 都用 OK 消息进行了应答，如图 6.20(b)所示。进程 4 接到第一个应答就知道它的工作已经结束了。它知道，有进程号比它大的进程将接管它的工作并成为协作者。它剩下的工作就是等着看谁将是获胜者(尽管此时它已经可以猜到结果了)。

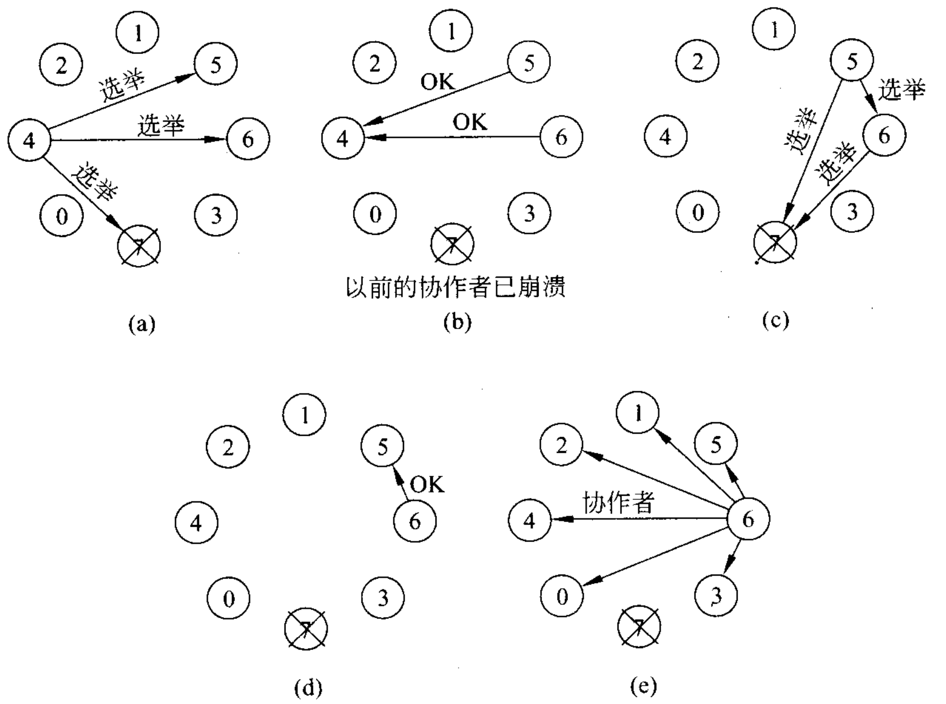


图 6.20 欺负选举算法

(a) 进程 4 主持一个选举；(b) 进程 5 和 6 进行响应，告诉进程 4 停止选举；(c) 进程 5 和 6 此时各自主持一个选举；(d) 进程 6 通知进程 5 停止选举；(e) 进程 6 获胜，并通知每个进程

在图 6.20(c)中，进程 5 和 6 都主持选举，每个进程都只将消息发送给比自己进程号大的进程。在图 6.20(d)中，进程 6 告诉进程 5 它将接管成为协作者。此时进程 6 知道进程 7 已经崩溃了，它(进程 6)将是获胜者。如果从磁盘或其他地方可以获得一些表明原有的协作者在哪里失效的状态信息，那么此时进程 6 就必须承担所需工作。当进程 6 准备好接管时，它向所有正在运行着的进程发送一个 COORDINATOR 消息。当进程 4 接收到这个消息后，它就可以继续当发现进程 7 崩溃了时它正要做的工作，但是这次是以进程 6 作为协作者了。这样，进程 7 的故障得到了处理，工作可以继续了。

如果进程 7 重新启动，它将向其他所有进程发送一个 COORDINATOR 消息，让它们服从自己的协调。

6.5.1.2 环算法

另一个选举算法是基于环(ring)的使用。不像其他一些环算法，该算法不使用令牌。假设进程按照物理或逻辑顺序进行了排序，那么每个进程就都知道它的后继者是谁了。当任何一个进程注意到协作者不工作时，它就构造一个带有它自己的进程号的 ELECTION 消息，并将该消息发送给它的后继者。如果后继者崩溃了，发送者沿着此环跳过它的后继者发送给下一个进程，或者再下一个，直到找到一个正在运行的进程。在每一步中，发送者都将自己的进程号加到该消息列表中，以使自己成为协作者的候选人之一。

最终，消息返回到发起此次选举的进程。当发起者进程接收到一个包含它自己进程号的消息时，它识别出这个事件。此时，消息类型变成 COORDINATOR 消息，并再一次绕环运行，向所有进程通知谁是协作者(成员列表中进程号最大的那个)以及新环中的成员都有谁。这个消息在循环一周后被删除，随后每个进程都恢复原来的工作。

图 6.21 示意了当进程 2 和 5 同时发现以前的协作者进程 7 崩溃了时，将会发生什么。这两个进程各自构造一个 ELECTION 消息，并且让它们相互独立地绕环运行。最后，两个消息都将绕环走过全程，进程 2 和 5 分别将它们转化为 COORDINATOR 消息，这两个消息拥有相同的成员，相互顺序也相同。两个消息再绕环一周后都被删除。有多余的消息循环没有害处，最多是花费了一点带宽，但这也不是浪费。

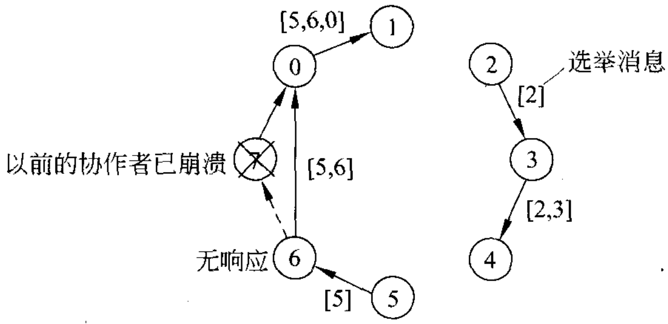


图 6.21 使用环的选举算法

6.5.2 无线系统环境中的选举算法

传统的选举算法基于的假设在无线系统环境中往往是不切实际的。例如，传统的选举算法假设消息传送是可靠的，网络的拓扑结构也是不会改变的。在大多数无线网络环境中，尤其是在移动自主网络中，这些假设都是错误的。

已开发的能在自主网络中工作的选举协议为数不多。(Vasudevan 等 2004)提出了一种能处理结点失效和网络分区的解决方法。这种方法的一个重要特性是，可以选举出最佳的领导者，而不是像在前面介绍的解决方法中那样只是随机的。其协议工作如下。为了简化起见，这里我们只是关注自主网络而忽略其中的结点是可以移动的。

来看一个无线自主网络。要选举一个领导者，网络中的任意结点(称为资源)可以通过往其紧邻结点(即与其在一条边上的结点)发送一条 ELECTION 消息来开始一个选举。当结点第一次接收到 ELECTION 消息时，它把该发送者指定为其父结点，然后把

ELECTION 消息发送给它的所有紧邻结点(其父结点除外)。当结点从某个结点(不是父结点)接收到一条 ELECTION 消息时,它只是确认这一接收。

当结点 R 指定结点 Q 为其父结点时,它把 ELECTION 消息转发给所有紧邻结点(结点 Q 除外),并在确认来自 Q 的 ELECTION 消息之前,等待来自其他结点的确认消息。这种等待的顺序很重要。首先要注意的是,已选择了父结点的邻结点会立即响应 R。更为特别的情况是,如果所有邻结点都已经有了父结点,R 就是一个叶结点,并可以快速响应 Q。在这样做的同时,它还向 Q 报告诸如其电池寿命和其他资源容量的信息。

随后,利用这些信息,Q 就可以将 R 的容量与其他下游结点的进行比较,并选择最合格的结点作为领导候选者。当然,Q 之所以发送一条 ELECTION 消息,是因为其父结点 P 也已经这么做了。反过来,当 Q 最终确认了前面由 P 发送的 ELECTION 消息时,它同样会把最合格的结点发送给 P。这样,源结点最终就知道哪个结点选作领导者最好,然后它就可以把这个信息广播给所有其他结点了。

在图 6.22 中解释了该过程。结点用 a～j 进行标记,同时还标记了它们的容量。结点 a 通过向结点 b～j 广播一个 ELECTION 消息来发起选举,如图 6.22(b)所示。然后,ELECTION 消息被传播给所有结点,直到如图 6.22(e)所示的情况为止,这里我们省略了由结点 f 和 i 发出的广播。然后,每个结点向其父结点报告具有最佳容量的结点,如图 6.22(f)所示。例如,当结点 g 从其孩子结点 e 和 h 接收到确认消息时,它就知道 h 是最佳结点,然后把[h, 8]传播给其父结点 b。最后,源结点就知道结点 h 是最佳领导者,并把这一信息广播给所有其他结点。

当发起了多个选举时,每个结点只能决定加入其中的一个。每个源结点用一个唯一的标识符来标记自己的 ELECTION 消息。结点只参与具有最高标识符的选举,同时停止参与其他选举。

只要稍作修改,该协议就可以在网络分区以及有结点加入和离开的情况中使用。详细信息请参见(Vasudevan 等 2004)。

6.5.3 大型系统中的选举算法

到目前为止我们所讨论的算法通常是应用在相对较小是分布式系统中。而且,这些算法只关注单个结点的选举。在很多情况下,需要选举多个结点,例如在点对点网络中的**超级点**(superpeer),我们在第 2 章中已介绍过了。在本节中,我们来特别关注一下超级点的选举问题。

要满足超级点的选举,(Lo 等 2005)列举了以下一些需求:

(1) 一般的结点访问超级点的延时要低;

(2) 超级点应平均地分布在覆盖网络中;

(3) 相对于覆盖网络中的所有结点,应有一部分预定义的超级点;

(4) 每个超级点无需为固定数目的一般结点服务。

幸运的是,在大多数点对点系统中,如果覆盖网络是结构化的(如基于 DHT 的系统),或者是随机非结构化的(如可以用基于 gossiping 的解决方法实现),这些需求的满足还是相对简单的。

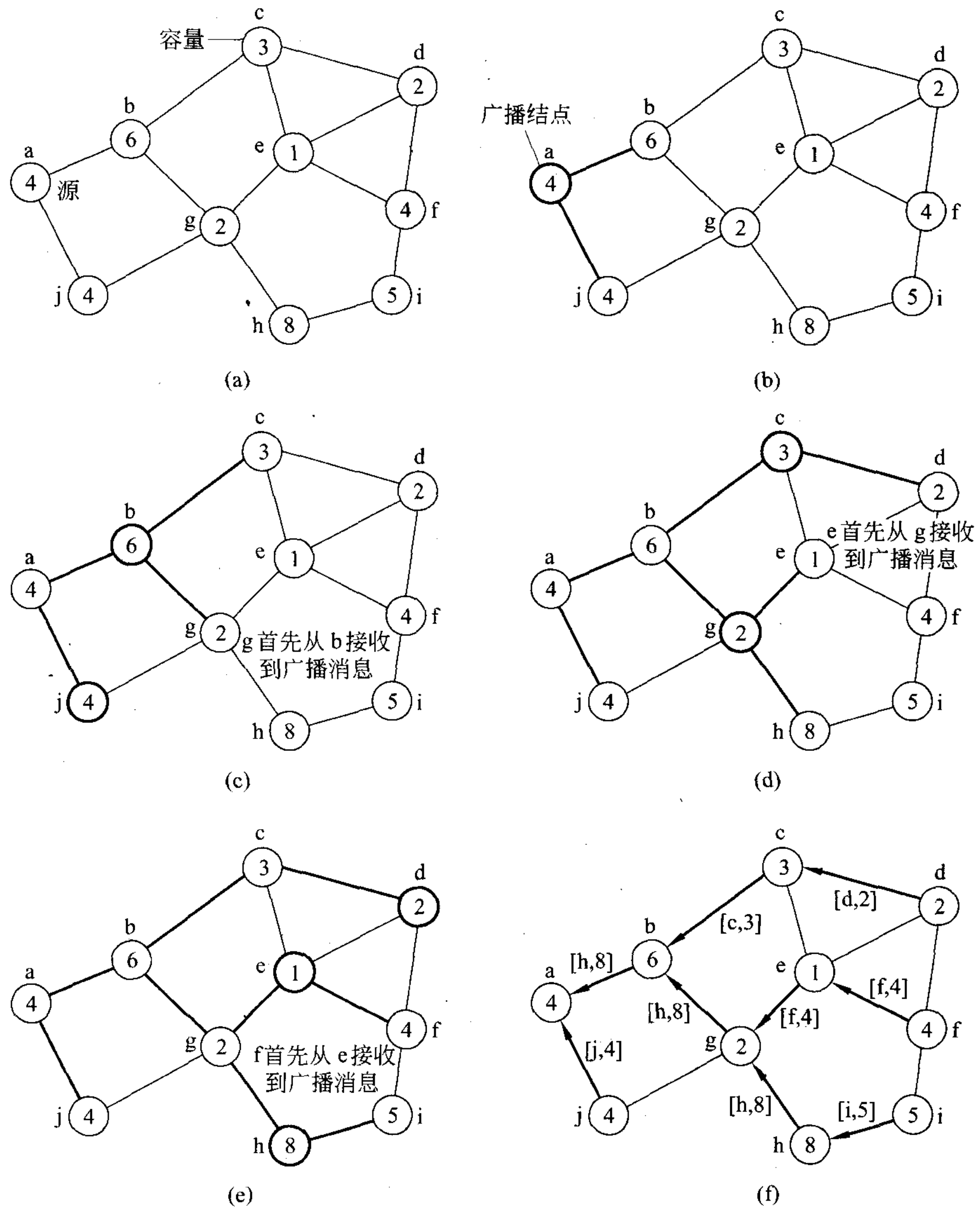


图 6.22 无线网络中的选举算法，其中 a 为源结点

(a) 初始网络；(b)～(e) 构建树的各个阶段；(f) 向源结点报告最佳结点

在基于 DHT 的系统中，基本思想是为超级点预留一部分标识符。回忆可知，在基于 DHT 的系统中，每个结点接收一个随机的且统一赋予的 m 位标识符。现在，假设预留前(即最左边的)k 位来标识超级点。例如，如果需要 N 个超级点，那么任意键值的前 $\lceil \log_2(N) \rceil$ 位可用来标识这些结点。

为了便于解释，我们假设有一个(小型的)Chord 系统，m＝8，k＝3。要查找负责特定键值 p 的结点，首先可以把查询请求路由给如下模式的结点：

```
p AND 11100000
```

然后把给结点视作超级点。注意，每个结点 id 可以通过查询

```
id AND 11100000
```

来看看该请求是否路由给了自己，以检查它是否是一个超级点。假设结点标识符统一地赋给了结点，可以看到，总共 N 个结点中平均有 $2^{k-m}N$ 个超级点。

一种完全不同的方法是在 m 维几何空间中定位结点，我们前面已经介绍过。在这种情况下，假设我们需要在整个覆盖网络中均匀地放置 N 个超级点。其基本思想很简单：总共 N 个令牌分布在 N 个随机选择的结点中。没有哪个结点可以拥有不止一个令牌。每个令牌表示一个推动力，推动另一个令牌向前移动。结果是，如果所有令牌发挥相同的推动力，那么它们就会相互之间移开，并均匀地分布在几何空间中。

这种方法要求结点含有的令牌了解其他令牌的信息。Lo 等提议使用 gossiping 协议，利用它，令牌的推动力就分散在整个网络中。如果某个结点发现总的推动力超过了某个阀值，那么该结点就向其合力的方向移动如图 6.23 所示。

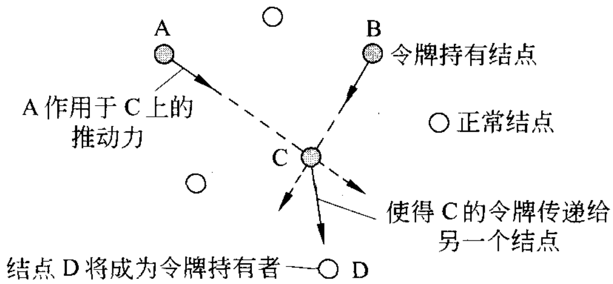


图 6.23 使用推动力在二维空间中移动令牌

当某个令牌被一个结点拥有了给定长的时间，该结点就会把自己提升为超级点。

本章小结

分布式系统中的进程同步问题与进程间通信有密切的关系。同步就是在适当的时刻做恰当的事情。分布式系统和计算机网络中的一个普遍问题是没有全局共享时钟。换句话说，不同机器上的进程都有自己的时间。

分布式系统中有很多同步化时钟的方法，但是所有方法本质上都是基于交换时钟值，同时考虑了发送和接收消息时所采用的时间。通信延迟的变化和对这些变化处理的方式很大程度决定了时钟同步化算法的准确性。

与这些同步化问题有关的是几何覆盖网络中定位结点。其基本思想是为每个结点赋给一个 m 维的坐标，这样，几何距离就可以用来精确地度量两个结点之间的延时。赋给坐标值的方法很像在 GPS 中用于确定位置和时间的方法。

许多情况下是不需要知道绝对时间的。重要的是不同进程中的相关事件以正确的顺序发生。Lamport 算法说明，通过引入逻辑时钟的概念，可以使得进程集就事件的正确顺序达成全局的一致。本质上，每个事件 e，例如发送或者接收消息，都被分配给一个全局唯一的逻辑时间戳 C(e)，这样，当事件 a 发生在事件 b 之前时，$C(a)<C(b)$。Lamport 时间戳可以扩展成向量时间戳：如果 $C(a)<C(b)$，那么我们就知道事件 a 在因果关系上发生在 b 之前。

一种重要的同步算法是分布式互斥。这些算法确保在分布式进程集中，每次至多一个进程可以访问共享资源。如果我们使用一个协作者来跟踪应该轮到谁来访问时，分布式互斥就可以容易地实现。还有完全的分布式算法，但是它们有缺点，它们更容易使通信和进程

发生故障。

进程间的同步化常常要求一个进程扮演协作者。在这些情况下,协作者不是固定的,需要分布式计算来决定谁将成为协作者。可通过选举算法来做这样的决定。选举算法主要用于协作者可能崩溃的情况。但是,它们还可以用于在点对点系统中选举超级点。

习 题

1. 请说出在 WWV 广播时间和分布式系统的处理器设置内部时钟之间,可以引入的延迟源(至少三种)。
2. 考虑分布式系统中的两台机器的行为。假设这两台机器的时钟每毫秒滴答 1000 次。但实际上只有一台是这样的,另一台每毫秒滴答 990 次,如果 UTC 每分钟更新一次,那么时钟的最大偏移量是多大?
3. 一种可以悄然进入分布式系统的现代设备是 GPS 接收器。请给出一些可以使用 GPS 信息的分布式应用示例。
4. 当某个结点要使其时钟与另一个结点的时钟同步化时,通常,一个较好的想法是还要把以前的度量考虑进去。为什么?并请给出这样一个示例。
5. 往图 6.9 中加入一个与消息 A 并发的新消息,即它既不发生在 A 的前面也不发生在 A 的后面。
6. 要使用 Lamport 时间戳实现全序多播,是不是每个消息都必须要被严格地确认?
7. 考虑一个消息只是以它们被发送的顺序被传递的通信层。给出一个不需要这种限制的例子。
8. 许多分布式算法需要使用协调进程。讨论一下,这样的算法实际上可以在什么程度上被看作为分布式的?
9. 在集中式互斥方法中(图 6.14),协作者在收到一个进程释放它独占访问的资源的消息后,通常给等待队列中的第一个进程授权。请给出协作者另一个可能的算法。
10. 请再考虑图 6.14。假设协作者崩溃了。这会必然导致整个系统瘫痪吗?如果不会,那么在什么情况下会呢?有什么方法避免这个问题的发生,使系统能够容忍协作者的崩溃?
11. Ricart 和 Agrawala 算法会有这样的问题:如果一个进程崩溃,并且没有对另一个请求访问资源的进程的请求作应答,没有应答意味着拒绝请求。我们建议所有的请求应当立即被应答,以便很容易地检测到崩溃的进程。是否存在一些情况,即使使用这种方法也还不足够?请讨论。
12. 如果我们假设该算法可以在支持硬件广播的 LAN 中实现,那么图 6.17 中的实体将如何改变?
13. 分布式系统可能有多个互相独立的资源。假设进程 0 要访问资源 A 而进程 1 要访问资源 B。Ricart 和 Agrawala 的算法会导致死锁吗?请解释原因。
14. 假设两个进程同时检测到协作者崩溃了,并且它们都使用欺负算法主持一个选举。这时将发生什么?
15. 在图 6.21 中,有两个 ELECTION 消息同时在循环。当对它们都没有什么不利影响时,杀掉一个将更好。设计这样一个算法,该算法在不影响基本的选举操作算法的情况完

成此项任务。

16. (**实验**)UNIX 系统提供了很多工具来使计算机保持同步，特别是把 crontab 工具()和各种同步化命令组合起来使用时，功能很强大。请配置一 UNIX 系统，使其本地时间的精确度保持在一秒以内。同样，配置一个自动备份工具，使得关键文件每隔 5 分钟就可以自动地传输到远程机器上。你的解决方法应确保在带宽有限时也能有效运行。

第7章

一致性和复制

分布式系统的一个重要问题是数据的复制。对数据进行复制一般是为了增强系统的可靠性或提高性能。实现数据复制的一个主要难题是保持各个副本的一致性。通俗地说，这意味着在更新一个副本时，必须确保同时更新其他副本；否则，数据的各个副本将不再相同。本章将深入讨论复制数据的一致性的真正含义，以及实现这种一致性的各种方法。

首先，我们概要地介绍复制的用途，以及它与可扩展性的关系。然后介绍什么是真正的一致性。一种重要的一致性模型是多个进程同时访问共享数据的情况。在这种情况下，根据什么进程希望何时读取和更新共享数据，同时知道其他进程正在访问该数据，就可以明确地表示一致性。

通常，在大规模分布式系统中有效地实现共享数据的一致性模型是很困难的。而且，在很多情况下可以采用较为简单的模型。它们也往往比较容易实现。以客户为中心的一致性模型就是其中一种。这种模型从一个单一的(可能是移动的)客户的角度来关注一致性。我们将在单独的一节中讨论这种以客户为中心的一致性模型。

一致性仅仅是问题的一个方面，我们还要考虑如何真正地实现一致性。这里主要有两个问题需要考虑。首先，我们关注的是副本的管理，这不仅要考虑到副本服务器的放置，还有考虑到内容是如何在这些服务器之间分发的。

第二个问题是如何保持多个副本的一致性。在大多数情况下，应用程序都要求数据保持很强的一致性。简单地说，这意味着数据更新必须快速地传播给每个副本。实现这种强一致性有多种不同的方法，这些方法将在单独的章节中进行讨论。另外，我们还将讨论一类特殊的一致性协议，即缓存协议。

7.1 概　　述

本节首先讨论进行数据复制的主要原因。我们将复制作为一种实现可扩展性的技术进行讨论，并指出一致性如此重要的原因。

7.1.1 进行复制的原因

进行数据复制主要出于两个目的：可靠性和性能。首先，数据复制可以提高系统的可

靠性。如果一个文件系统已经实现数据复制,那么当一个副本被破坏后,文件系统只需转换到另一个数据副本就可以继续运行下去。同样,通过维护多个副本,系统可以更好地防止数据破坏。例如,假设一个文件存在三个副本,而且每个读写操作都在各个副本上执行。这样,可以保护数据不会因为一个失败的写操作而受到破坏,因为此时至少两个副本的值是正确的。

进行数据复制的另一个目的是提高性能。当分布式系统需要在服务器数量和地理区域上进行扩展时,复制对于提高性能是相当重要的。例如,当需要访问由单一服务器管理的数据的进程数不断增加时,系统就需要进行服务器数量上的扩充。在这种情况下,通过对服务器进行复制,随后让它们分担工作负荷,就可以提高性能。

地理区域上的扩展也可能需要进行数据复制。其基本思想是,如果在使用数据的进程附近放置一份该数据的副本,那么进程访问数据所花费的时间将减少。对于这个进程来说,性能就相对地提高了。但这个示例也说明了复制给性能带来的益处是难以评估的。虽然一个客户进程可能获得较好的性能,但是这也可能导致为维持所有副本的更新而消耗了更多的网络带宽。

如果复制有助于提高可靠性和性能,那么谁还会反对呢?然而,进行数据复制是需要付出代价的。复制所带来的问题是多个副本可能导致一致性方面的问题。一旦某个副本被修改了,那么它将不同于其他所有的副本。因此,必须对所有的副本进行同样的修改以确保一致性。执行这些修改的确切时间和方式决定了复制代价的大小。

为了便于理解这个问题,来看看改善 Web 页面访问时间的示例。如果不采用特别的方法,从远程 Web 服务器获取一个页面有时甚至需要几秒的时间才能完成。为了提高性能,Web 浏览器通常都在本地存储一个以前获取的该 Web 页面的副本(也就是说,Web 浏览器对 Web 页面进行**缓存**(cache))。如果用户需要再次访问那个页面,浏览器会自动返回本地副本。这样,用户所觉察到的访问时间是非常短的。然而,如果用户总是想获得最新版本的页面,他可能就会不那么走运了。问题在于,如果此时对页面进行了更新,而这些更新还没有传播到被缓存的副本,那么缓存的副本就是过时的了。

为了解决这个问题,一种方法是禁止浏览器保存本地副本,由服务器完全负责复制事宜。然而,如果用户附近没有数据副本,这种方法会导致访问时间的延长。另一种方法是由 Web 服务器标识出无效的缓存副本并对它们进行更新,但是这种方法要求服务器跟踪所有的高速缓存,并向它们发送消息。这可能会降低服务器的整体性能。我们将在下面接着讨论性能与可扩展性的问题。

7.1.2 作为扩展技术的复制

为了提高性能,复制和缓存作为扩展技术得到了广泛的应用。可扩展性问题往往以性能问题的形式出现。将数据和对象的副本放置在使用它们的进程的附近,可以减少访问时间,提高性能,从而解决可扩展性问题。

可能要一个付出的代价是使副本保持为最新的数据要求更多的网络带宽。例如,一个进程 P 每秒访问一个本地副本 N 次,而这个副本自身每秒更新 M 次。假设每次更新都完全刷新本地副本的原有版本。如果 $N \ll M$,也就是说,访问与更新的比率很低,那么我们所

面临的情况是，很多本地副本的更新版本从未被P访问过，使得为这些版本更新而付出的网络通信毫无用处。在这种情况下，最好不要在P附近安装本地副本，或者应该采用其他更新副本的策略。后面还会继续讨论这一问题。

然而，一个更为严重的问题是，保持多个副本间的一致性本身就可能存在严重的可扩展性问题。直观地看，当多个副本总保持相同的时候，这些副本的集合是一致的。这意味着在任何副本上执行读操作都将返回相同的结果。因此，在一个副本上执行更新操作时，无论这一操作是在哪个副本上启动或执行的，这一更新操作都应该在后续操作发生前传播到所有副本。

这种类型的一致性有时不规范地（也不准确地）称为强一致性，如同所谓的同步复制所提供的一致性（我们将在下一节中给出一致性的准确定义，并引入一系列一致性模型）。其关键思想是以单个的原子操作或事务的形式在所有副本上执行更新。不幸的是，当这种原子操作涉及到大量的、可能广泛散布在大型网络中的副本，同时要求原子操作快速完成时，实现这些操作是相当困难的。

其中的困难来自于需要同步化所有副本这一事实。从本质上说，这意味着所有副本首先需要就本地执行更新的确切时间达成共识。例如，副本可能需要使用 Lamport 时间戳来决定执行操作的全局顺序，或由一个协调器来分配这样一个顺序。全局同步需占用大量通信时间，特别是副本散布于广域网时更是如此。

现在我们处于进退两难的境地。一方面，可扩展性问题可以通过采用复制和缓存技术得到缓解。而另一方面，保持所有副本的一致性通常要求全局同步，而全局同步必然造成性能的严重下降。这似乎有些得不偿失。

在很多情况下，唯一可行的解决方法是放宽在一致性方面的限制。也就是说，如果可以放松"更新必须以原子操作的方式执行"这一要求，就可以避免进行（瞬间的）全局同步，从而可以获得较好的性能。所付出的代价是各个副本可能并不总是相同的。事实表明，一致性可被放宽的程度主要取决于复制数据的访问和更新模式，同时还取决于这些数据的用途。

在以下几节中，我们将先讨论一系列一致性模型。这些模型提供一致性的准确定义。然后，我们将讨论通过分发和一致性协议实现一致性模型的不同方法。有关一致性和复制的不同分类方法可以参见（Gray 等 1996；Wiesmann 等 2000）。

7.2 以数据为中心的一致性模型

通常，总是在讨论共享数据的读操作和写操作时讨论一致性问题。这些共享数据是通过分布式共享存储器、分布式共享数据库或分布式文件系统实现的。在本节中，我们将使用更广义的术语，即**数据存储**（data store）。数据存储可以物理地分布在多台机器上。特别是，假设每个可以访问数据存储中的数据的进程都有整个数据存储的一个本地或邻近的副本，写操作将传播到其他副本，如图 7.1 所示。一个更改数据的数据操作被归类为写操作，否则归类为读操作。

一致性模型（consistency model）实质上是进程和数据存储之间的一个约定。即，如果进程同意遵守某些规则，那么数据存储将正常运行。正常情况下，一个进程在一个数据项上

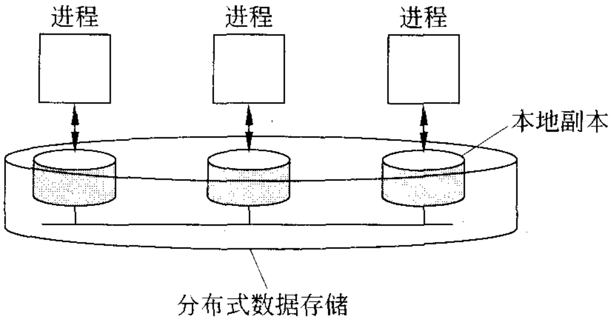


图 7.1　逻辑数据存储的一般组织结构，该数据存储在物理上是分布的，并被复制到多个进程中

执行读操作时，它期待该操作返回的是该数据在其最后一次写操作之后的结果。

在没有全局时钟的情况下，精确地定义哪次写操作是最后一次写操作是十分困难的。作为替代的方法，我们需要提供其他的定义，因此产生了一系列的一致性模型。每种模型都有效地限制了在一个数据项上执行一次读操作所应返回的值。正如可以预料到的，带有较少限制的模型比较容易使用，而带有较多限制的模型使用起来则比较困难。当然，在另一方面，容易使用的模型的执行效果不如较复杂的模型那样好。这就是生活。

7.2.1　持续一致性

从前面的讨论可以清楚地看出，对复制数据问题没有最好的解决方法。复制数据会导致一致性问题，这用一般方法不能有效地解决。除非放宽对一致性的限制才有希望获得有效的解决方法。但是也没有放宽一致性的通用标准：哪些东西可以容忍很大程度上是取决于应用程序。

有多种不同的方法来为应用程序指定它们能容忍哪些不一致性。（Yu 和 Vahdat 2002）给出了一种通用方法，为定义不一致性区分了三个相互独立的坐标轴：副本之间的数值偏差、副本之间新旧程度的偏差以及更新操作顺序的偏差。他们称这些偏差形成了**持续一致性**（continuous consistency）的范围。

其数据具有数值语义的应用程序可以用按数值偏差来度量不一致性。一个明显的示例是股票市场价格记录的复制。在这种情况下，应用程序可能会指定两个副本的偏差不能超过 0.02 美元，这就是**绝对数值偏差**（absolute numerical deviation）。也可以指定**相对数值偏差**（relative numerical deviation），表示两个副本之间的差别不能超过多少（例如 0.5%）。

数值偏差可以这样来理解：已应用于给定的副本但还没有应用于其他副本的更新数目。例如，一个 Web 缓存可能还没有得到 Web 服务器执行的一批操作。在这种情况下，数值的相关偏差又称为权重。

新旧偏差与副本最近一次更新有关。对某些应用程序来说，只要副本提供的旧数据不是太旧，它是可以容忍的。例如，天气预报通常要滞后一段时间（如几个小时）。在这种情况下，主服务器会定期地接收更新信息，但在一段时间内只给副本传播一次更新信息。

最后，在有些应用程序中，只要可以界定副本之间的差异，就允许不同的副本采用不同的更新顺序。这些更新的一种方法是，暂时性地把这些更新应用到一个本地副本，等待所有

副本达成全局一致。因此，可能需要回滚某些更新，在成为永久更新之前，以不同的顺序进行更新。直观上来看，顺序偏差比另外两种一致性偏差更难掌握。我们将给出一些示例来阐述清楚。

一致性单元

为了定义非一致性，Yu 和 Vahdat 引入了**一致性单元**（consistency unit，简称为 conit）。一致性单元表示的是其一致性可以度量的单元。例如，在股票交易的示例中，就可以把表示单个股票的记录定义为一个一致性单元。另一个示例就是单个的天气预报。

为了给出一致性单元的一个示例，同时解释数值偏差和顺序偏差，请看图 7.2 所示的两个副本。每个副本 i 维护一个二维向量时钟 VC_i，就像我们在第 6 章所介绍的那样。我们使用(t, i)来表示副本 i 在逻辑时间 t 中执行的一个操作。

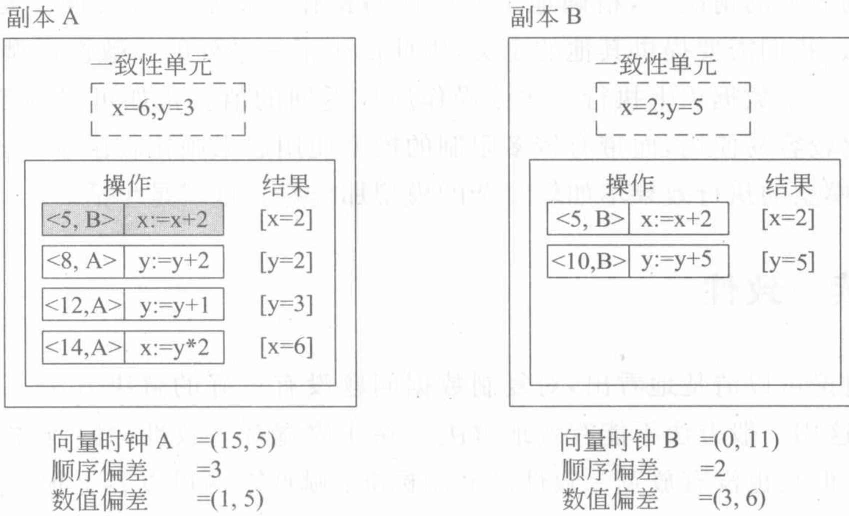


图 7.2 阐述一致性偏差的示例（摘自（Yu 和 Vahdat 2002））

在这个示例中，我们可以看到两个副本作用在一个含有数据项 x 和 y 的一致性单元上。这两个变量都假设被初始化为 0。副本 A 从副本 B 接收操作

5, B: x←x+2

并使其为永久性的（也就是说，该操作被提交给 A，且不能回滚）。副本 A 有三个暂时的更新操作(8, A)、(12, A)和(14, A)，这给它带来的顺序偏差为 3。还要注意的是，由于最后的操作是(14, A)，A 的向量时钟变成了(15, 5)。

A 还没有看到来自 B 的唯一操作是(10, B)，使其数值偏差为 1。这本示例中，该偏差的权重可以表示为 A 的（已提交）值 x 和 y，与 A 还没有接收到的来自 B 的操作产生的结果之间的最大差分。A 的已提交值是(x, y)=(2, 0)，而（A 还没有接收到的）B 操作产生了 y=5 的差分。

同样的推理可以证明，B 有两个暂时的更新操作(5, B)和(10, B)，这意味着它的顺序偏差为 2。因为 B 没有接收到来自 A 的操作，因此其向量时钟变成了(0, 11)。数值偏差为 3，总权重为 6。

注意，这里需要在维护细颗粒的一致性单元与粗颗粒的一致性单元之间进行平衡。如果一致性单元表示的是大量数据，如一个完整的数据库，那么为所有数据的更新汇集成一个

一致性单元。因此，迟早会使得副本处于非一致的状态。例如，在图 7.3 中，假设两个副本之间相差不超过一个输出更新。在这种情况下，当图 7.3(a)的数据项更新了第一个副本，那么第二个也需要被更新。如果如图 7.3(b)所示的那样选择更小的一致性单元，那么情况就不是这样了。这里副本仍认为是最新的。这个问题重要的部分原因是，包含在一致性单元中的数据项是完全独立使用的，这种情况被称为**虚假共享**(falsely share)一个一致性单元。

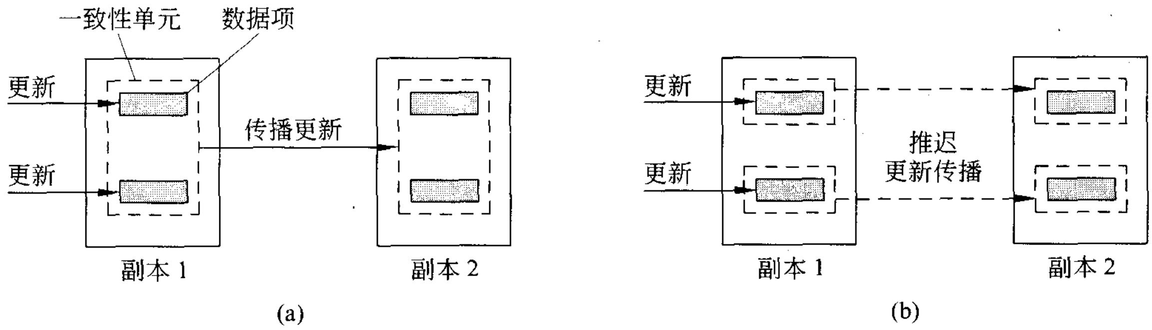


图 7.3　为一致性单元选择适当的粒度

(a) 两次更新后才进行更新传播；(b) 无需更新传播

但是，使得一致性单元太小并不是好主意，原因很简单，这样的话，需要管理的一致性单元总数也增加了。换句话说，这里需要考虑到管理一致性单元的费用。管理费用可能会对整体性能有负面影响，因此需要给予考虑。

从概念上来说，一致性单元为满足一致性需求提供了一种很吸引人的方法，但在把它们放到实际使用中之前，还需要处理两个重要的问题。首先，为了确保一致性，需要一些协议。关于持续一致性的协议将在本章后面讨论。

第二个问题是，持续程序开发人员必须为他们的应用程序指出一致性需求。实践表明，获得这种需求可能会非常困难。编程人员往往不熟悉处理复制问题，更不要说让他们去理解一致性的详细信息了。因此，编程接口必须是简单且容易理解的。

可以把持续一致性实现为一种工具包，对编程人员来说，这种工具包就像是一个函数库，他们可以把它链接到他们的应用程序中去。可以把一个一致性单元声明为只是一个数据项的一次更新。例如，下面伪代码段：

```
AffectsConit(ConitQ, 1);
appen messege m to queue Q;
```

表示把一个消息附加到队列 Q 中，其中队列 Q 属于一个名为 ConitQ 的一致性单元。同样，现在也可以如下声明与一致性单元有关的操作：

```
DependsOnConit(ConitQ, 4, 0, 60);
read message m from head of queue Q;
```

在这种情况下，对 DependsOnConit 的调用指明，数值偏差、顺序偏差和新旧程度偏差必须分别限制在 4、0 和 60(秒)。这可以解释为，其他副本最多只能有 4 个未接收到的更新操作，无暂时的本地更新，在 60 秒钟内必须检查 Q 的本地副本的新旧程度。如果这些需求没有完全满足，底层中间件将把 Q 的本地副本置于可以执行读操作的状态。

7.2.2 一致的操作顺序

过去的几十年以来，除了持续一致性以外，人们对以数据为中心的一致性模型做了大量工作。一种重要的模型来自并发程序设计。在并行和分布式计算中，面临着这样一个事实：多个进程需要同时共享资源并访问这些资源，研究人员寻求描述当共享资源被复制时的并发访问的语义。这就至少产生了一种重要的、被广泛使用的一致性模型。下面，我们来关注顺序一致性，我们还将讨论一种更弱的变体，即因果一致性。

7.2.2.1 顺序一致性

下面我们将使用一种特殊的表示法，沿一个时间轴来绘制一个进程的操作。时间轴总是水平绘制的，时间从左到右增长。符合

$W_i(x)a$ 和 $R_i(x)b$

分别表示进程 P_i 把数值 a 写入数据项 x，以及进程 P_i 从数据项 x 读取数据后返回数值 b。假设每个数据项初始时为 NIL。当不会混淆是哪个进程访问数据时，我们将把 W 和 R 的索引省略掉。

图 7.4 所示的示例往数据项 x 写入数据，把数据项 x 的值修改为 a。注意，原则上，操作 $W_1(x)a$ 首先在 P_1 的本地副本上执行，然后传播给其他本地副本。在这个示例中，P_2 随后读取值 NIL，然后（从其本地副本中）读取值 a。这里我们可以看到，把 x 的更新传播给 P_2 花费了一些时间，这是可以接受的。



图 7.4 作用于相同数据项上的两个进程的行为

顺序一致性（sequential consistency）是一种重要的以数据为中心的一致性模型。它是由 Lamport(1979) 在解决多处理器系统的共享存储器时首次提出的。通常，数据存储满足以下条件时，称为是顺序一致的：

> 任何执行结果都是相同的，就好像所有进程对数据存储的读、写操作是按某种序列顺序执行的，并且每个进程的操作按照程序所制定的顺序出现在这个序列中。

以上定义的意思是，当进程在多台（可能）不同机器上并发运行时，任何读、写操作的有效交叉都是可接受的行为，但是所有进程都看到相同的操作交叉。注意，这里并未提及时间，也就是说，这里并未提及“最近一次”对一个对象的写操作。在这里，进程可以“看到”所有进程的写操作，但是只能看到它自己的读操作。

从图 7.5 可以看出，时间不起作用。假设有 4 个进程对同一数据项 x 进行操作。图 7.5(a) 中的进程 P_1 对数据项 x 执行 W(x)a 写操作，随后（绝对时间），进程 P_2 也执行了一次写操作，将 x 的值设置为 b。然而，进程 P_3 和 P_4 先读到值 b，然后才读到值 a。换句话说，进程 P_2 的写操作发生在进程 P_1 的写操作之前。

而图 7.5(b) 违背了顺序一致性，因为不是所有的进程看到了相同的写操作交叉。特别是对进程 P_3 而言，数据项好像首先变为 b，然后变为 a。而进程 P_4 却得到数据项的最终值 b。

P_1: W(x)a				
P_2:	W(x)b			
P_3:		R(x)b		R(x)a
P_4:			R(x)b	R(x)a

(a)

P_1: W(x)a				
P_2:	W(x)b			
P_3:		R(x)b		R(x)a
P_4:			R(x)a	R(x)b

(b)

图　7.5

(a) 顺序一致的数据存储；(b) 非顺序一致的数据存储

为了更具体地理解顺序一致性的含义，让我们来看看图 7.6（Dubois 等 1988）所示的三个并发执行的进程 P_1、P_2 和 P_3。在这个例子中，数据项是三个整型变量 x、y 和 z，它们存储在顺序一致（也可能是分布的）的共享数据存储中。假设每个变量的初始值都是 0。在本例中，赋值语句与写操作对应，而显示语句与该时刻它所带两个参数的读操作对应。假定所有语句都是不可分割的。

进程 P_1	进程 P_2	进程 P_3
x ← 1;	y ← 1;	z ← 1;
print(y, z);	print(x, z);	print(x, y);

图 7.6　三个并发执行的进程

各种交叉执行序列都是可能的。对于 6 个独立的语句而言，将有 720（即 6!）种可能的执行序列，其中某些序列可能违背了程序顺序。假设 120（即 5!）种序列以 x←1 开始，一半的序列（即 60 种）在 y←1 之前执行了 print(x,z)，那么这些序列就违背了程序顺序。然后这其中的一半（即 30 种）序列在 z←1 之前执行了 print(x,y)，也违背了程序顺序。这样，只有 120 种的四分之一的序列（即 30 种）是有效的。以 y←1 开始的序列有 30 种有效序列，以 z←1 开始的有效序列也有 30 种。因此，总共有 90 种有效的执行序列。图 7.7 列出了其中 4 种有效序列。

在图 7.7(a) 中，三个进程以如下顺序运行：首先运行 P_1，然后运行 P_2，最后运行 P_3。而另外三个示例则说明了虽然在时间上不同但同时有效的语句的交叉。三个进程都输出两个变量的值。因为每个变量可取的唯一值都是初始值（0）或所赋的值（1），所以每个进程生成一个两位的字符串。Prints 后的数字是显示在输出设备上的实际输出。

如果将 P_1、P_2 和 P_3 的输出按照输出顺序连接起来，那么所得到的 6 位字符串说明了每种语句交叉的特点，如图 7.7 所示的 Signature（签名）字符串。下面我们将使用该字符串（而不是输出结果）来表示语句的执行顺序。

(a)	(b)	(c)	(d)
x ← 1;	x ← 1;	y ← 1;	y ← 1;
print(y,z)	y ← 1;	z ← 1;	x ← 1;
y ← 1;	print(x,z);	print(x,y);	z ← 1;
print(x,z);	print(y,z);	print(x,z);	print(x,z);
z ← 1;	z ← 1;	x ← 1;	print(y,z);
print(x,y);	print(x,y);	print(y,z);	print(x,y);
Prints: 001011	Prints: 101011	Prints: 010111	Prints: 111111
Signature: 001011	Signature: 101011	Signature: 110101	Signature: 111111

图 7.7　图 7.6 中进程的 4 个有效执行顺序。垂直轴是时间轴

不是所有 64 个签名模式都是允许的。例如,000000 就是不允许的,因为它意味着显示语句在赋值语句之前执行,这违反了语句必须按程序顺序执行的要求。另一个例子是 001001。前两位为 00,意味着当 P_1 执行显示语句时,b 和 c 的值都是 0。只有 P_1 在 P_2 或 P_3 启动之前执行这条语句才会出现这种情况。接下来的两位为 10,意味着 P_2 必须在 P_1 启动后而 P_3 仍未启动前执行。最后两位为 01,意味着 P_3 必须在 P_1 启动前完成。而我们已经看到 P_1 必须先执行,所以,001001 也是不允许的。

简而言之,90 个不同的有效语句顺序产生各种各样不同的程序结果(虽然少于 64 种),这些都是满足顺序一致性假设前提的结果。进程与分布式的共享数据存储间的协议是,进程必须只接受有效的程序结果。换句话说,进程必须只将图 7.7 所示的 4 种结果以及所有其他有效结果视为正确答案。如果出现其中任何一种结果,进程都必须正常地运行。仅对其中部分结果可以运行,而对另一些结果不能运行的程序违背了数据存储的协议,是不正确的。

7.2.2.2 因果一致性

因果一致性(causal consistency)模型(Hutto 和 Ahamad 1990)表示的是一种弱化的顺序一致性模型,因为它将具有潜在因果关系的事件和没有因果关系的事件区分开来。在前一章讨论向量时间戳时,我们已经遇到了因果关系的问题。如果事件 B 是由事件 A 引起的,或受到事件 A 的影响,那么因果关系必然要求其他每个人先看到事件 A,再看到事件 B。

考虑一个分布式数据库的示例。假设进程 P_1 对数据项 x 执行了写操作。然后,进程 P_2 先读取 x,然后对 y 执行写操作。这里,对 x 的读操作和对 y 的写操作具有潜在因果关系,因为 y 的计算可能依赖于 P_2 所读取的 x 的值(也就是,P_1 所写的值)。

另一方面,如果两个进程自发且同时地对两个不同的数据项执行写操作,这两个事件就不具备因果关系。无因果关系的操作称为是**并发的**(concurrent)。

如果数据库是因果一致的,那么它必须服从以下条件:

> 所有进程必须以相同的顺序看到具有潜在因果关系的写操作。不同机器上可以以不同的顺序看到并发的写操作。

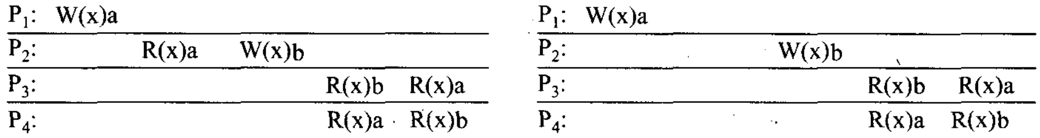
图 7.8 是一个因果一致性的示例。这里的事件顺序是因果一致的存储所允许的,但该顺序是顺序一致的存储或严格一致的存储所禁止的。值得注意的是,写操作 $W_2(x)b$ 和 $W_1(x)c$ 是并发的,所以没有必要要求所有进程以相同的顺序看到它们。

P_1:	W(x)a			W(x)c		
P_2:		R(x)a	W(x)b			
P_3:		R(x)a			R(x)c	R(x)b
P_4:		R(x)a			R(x)b	R(x)c

图 7.8 因果一致性存储所允许的但顺序一致性存储不允许的顺序

现在,我们看看另一个示例。图 7.9(a)中,$W_2(x)b$ 潜在地依赖于 $W_1(x)a$,这是因为 b 可能是使用 $R_2(x)a$ 所读取的值进行计算的结果。两个写操作是因果相关的,所以所有进程必须以相同的顺序看到这两个操作。因此,图 7.9(a)是不正确的。但是,图 7.9(b)删除了读操作,所以 $W_1(x)a$ 和 $W_2(x)b$ 现在是并发的写操作了。因果一致的存储并不要求并

发的写操作是全局有序的，所以图 7.9(b)是正确的。图 7.9(b)反映了顺序一致性存储所不能接受的一种情况。



(a)　　　　(b)

图　7.9

(a) 违背因果一致存储的顺序；(b) 符合因果一致存储顺序

实现因果一致性要求跟踪哪些进程看到了哪些写操作。这就意味着必须构建和维护一张记录哪些操作与其他哪些操作有关的关系图。一种实现方法是使用前一章所讨论的向量时间戳。我们将在本章后面再继续讨论使用向量时间戳捕获因果关系的方法。

7.2.2.3　分组操作

顺序一致性和因果一致性是在读写操作层面上定义的。这种粒度层是有历史原因的：这些模型最初是为了共享内存的多处理器系统开发的，在硬件层面上真正实现的。

这些一致性模型的粒度在很大情况下与应用程序提供的粒度并不匹配。我们可以看到，共享数据的程序之间的并发性往往是通过互斥和事务的同步化机制来控制的。在程序层面的读写操作包含在 ENTER_CS 和 LEVEL_CS 两个操作中了，其中 CS 表示的是临界区(critical section)。正如在第 6 章中所介绍的那样，进程间的同步化是通过这两个操作来完成的。在分布式数据存储中，这意味着已成功运行 ENTER_CS 的进程可以确保在它的本地存储中的数据是最新的。此时，它就可以在其本地数据存储中安全地运行一系列的读写操作，然后通过调用 LEVEL_CS 封装起来。

基本上，在程序中由一系列读写操作所作用的数据是被保护的，不让并发地访问，从而不会有该读写操作系列作为一个整体所产生的结果之外的其他东西。封装在一起的读写操作系列又分成了自动运行的单元，从而产生了粒度层。

为了实现这一点，我们需要有关于 ENTER_CS 和 LEVEL_CS 操作的精确语义。这些语义可以以共享的**同步化变量**(synchronization variable)来描述。有不同的方法来使用这些变量。我们将介绍常用的方法，其中的每个变量具有某些相关的数据，这些数据合起来就是共享数据的完整集。我们采用的约定是，当某个进程进入临界区时，它必须获得相关的同步化变量，同样，当它离开临界区时，要释放这些变量。注意，进程的临界区中的数据可能与不同的同步化变量相关。

每个同步化变量都有其当前拥有者，即最后获得它的进程。该拥有者可以反复进入和退出临界区，无需在网络上发送任何消息。如果某个进程现在不拥有某个同步化变量但想要获得它，该进程就必须往该变量的当前拥有者发送一个消息，请求拥有权以及与该同步化变量相关的数据的当前值。也有可能在非互斥模式中，多个进程同时拥有某个同步化变量，它们可以读取但不能写有关的数据。

现在，我们就可以提出如下应满足的标准(Bershad 等 1993)：

1. 在一个进程对被保护的共享数据的所有更新操作执行完之前，不允许另一个进程执行对同步化变量的获取访问。
2. 如果一个进程对某个同步化变量正进行互斥模式访问，那么其他进程就不能拥有该同步化变量，即使是非互斥模式也不行。
3. 某个进程对某个同步化变量的互斥模式访问完成后，除非该变量的拥有者执行完操作，否则任何其他进程对该变量的下一个非互斥模式访问也是不允许的。

第一个条件表示，当一个进程获得拥有权后，这种拥有权直到所有被保护的数据都已更新为止。换句话说，对被保护数据的所有远程修改都是可见的。

第二个条件表示，在更新一个共享数据项之前，进程必须以互斥模式进入临界区，以确保不会有其他进程试图同时更新该共享数据。

第三个条件表示，如果一个进程要以非互斥模式进入临界区，必须首先与该同步化变量的拥有者进行协商，确保临界区获得了被保护共享数据的最新副本。

图 7.10 显示了一个名为**入口一致性**(entry consistency)的示例。在这个示例中，不是作用在整个共享数据上，而是把锁与每个数据项关联起来。在这种情况下，P_1 获得了 x，修改 x 一次，然后在获得 y。进程 P_2 获得了 x，但没有获得 y，因此它可以从 x 读取值 a，但如果读取 y 则为 NIL。因为进程 P_3 首先获得了 y，所以当 P_1 释放 y 时，P_3 可以读得值 b。

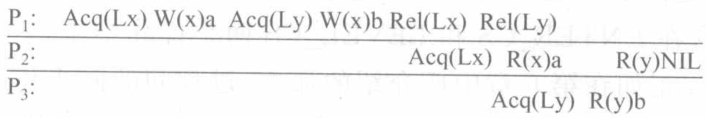


图 7.10 入口一致性的合法事件顺序

入口一致性的编程问题之一是，正确地把数据与同步化变量相关联。一种简单的方法是显式地告诉中间件将要访问哪个数据，就像声明哪个数据库表将被一个事务修改一样。在基于对象的方法中，我们可以隐式地把唯一的同步化变量与每个已声明的对象相关联，有效地串行化对该对象的所有调用。

7.2.2.4 一致性与相干性

至此，有必要澄清两个很有关联的概念的区别。到此为止，我们所讨论的模型都是用于处理在一个数据项集上进行读写操作的进程。**一致性模型**(consistency model)描述的是多个进程并发地作用于数据的数据集。如果该数据集遵守该模型描述的规则，那么就称之为一致性的。

数据一致性关注的是数据项集，而**相干性模型**(coherence model)关注的只是单个数据项(Cantin 等 2005)。在这种情况下，我们假设某个数据项在多个地方被复制，当各个副本遵守相干模型定义的规则时，我们称之为相干的。一种常见的模型是顺序一致性模型，现在只应用于单个数据项。它表示，在并发写数据的情况下，所有进程最终都将看到相同的更新顺序。

7.3 以客户为中心的一致性模型

上一节讲述的一致性模型的目的在于从数据存储的角度提供系统级别的一致性。一个重要的假设是并发进程可能同时更新数据存储，而在发生这种并发更新时，必须提供一致性。例如，在基于对象的入口一致性的情况下，数据存储保证向调用对象的进程提供该对象的一个副本。这个副本可以反映出对该对象所做的所有改变，而这些改变可能是其他进程做出的。在调用的过程中，也可以保证其他的进程不会干扰调用，也就是说，向调用进程提供的是独占性访问。

维持共享数据的顺序一致性的同时能够处理共享数据上的并发操作，对于分布式系统而言是非常重要的。由于性能原因，可能只有当进程使用诸如事务处理和锁一类的同步机制时，系统才能保证顺序一致性。

本节中，我们将讨论一类特殊的分布式数据存储。这里讨论的数据存储的特点是：不会出现同时发生的更新操作，或者当出现同时发生的更新操作时，可以容易地解决它们。大部分操作是读取数据的操作。我们将讨论的数据存储提供一种很弱的一致性模型，称为最终一致性。通过引入特殊的以客户为中心的一致性模型，事实证明，可以以一种开销相对低的方法隐藏许多不一致性。

7.3.1 最终一致性

进程实际以并发方式操作的程度与所需保证的一致性的程度可能不同。并发操作只以有限的形式出现的示例很多。例如，在许多数据库系统中，大多数进程几乎从不执行更新操作，而只从数据库读取数据，有时只有一个或少数几个进程会执行更新操作。那么，现在的问题是应该以多快的速度进行更新操作，才能使更新对只读进程有效。

另一个示例是世界范围的命名系统，例如 DNS。DNS 名称空间被划分成域，每个域分配一个命名授权机构，由该命名授权机构担当这个域的所有者。只有授权机构才被允许更新名称空间中它所负责的部分。因此，这个系统根本不会出现两个更新相同数据的操作所导致的冲突(即，**写-写操作冲突**(write-write conflict))。唯一需要处理的情况是**读-写操作冲突**(read-write conflict)。事实证明，这种情况通常允许以懒惰方式传播更新操作，也就是说，读进程只有在更新操作执行一段时间后才能看到这个更新操作。

还有一个示例是 WWW。实际上，在所有的情况下，Web 页面是由单一的授权机构更新的，例如，Web 站点管理员或页面的实际所有者。通常情况下，不存在需要解决的写-写操作冲突的问题。但是，为了提高效率，通常配置浏览器和 Web 代理在本地高速缓存中保存一份已下载页面，并在用户下次请求该页面时返回高速缓存中的副本。两种类型的 Web 高速缓存都存在一个重要问题，那就是它们可能返回过时的 Web 页面。也就是说，与实际可从 Web 服务器获得的页面相比，向客户返回的高速缓存页面是一个较旧版本的页面。事实证明，许多用户认为这种不一致性是可接受的。

这些示例可以看作是能容忍相对较高程度的不一致性的(大规模的)分布式复制的数据

库的案例。它们的共同之处是，如果在一段很长的时间内没有更新操作，那么所有的副本将逐渐地成为一致的。这种形式的一致性称为**最终一致性**(eventual consistency)。

因此，满足最终一致性的数据存储具有以下属性：没有更新操作时，所有副本逐渐成为相互完全相同的副本。最终一致性实际上只要求更新操作被保证传播到所有副本。如果假设只有一小部分进程可以执行更新，那么写-写操作冲突就相对比较容易解决了。因此，最终一致性的实现通常开销比较小。

只要客户总是访问同一副本，最终一致的数据存储就会工作得很好。但是，当客户访问不同的副本时，问题就出现了。图7.11所示的移动用户访问分布式数据库的示例较好地描述了这一问题。

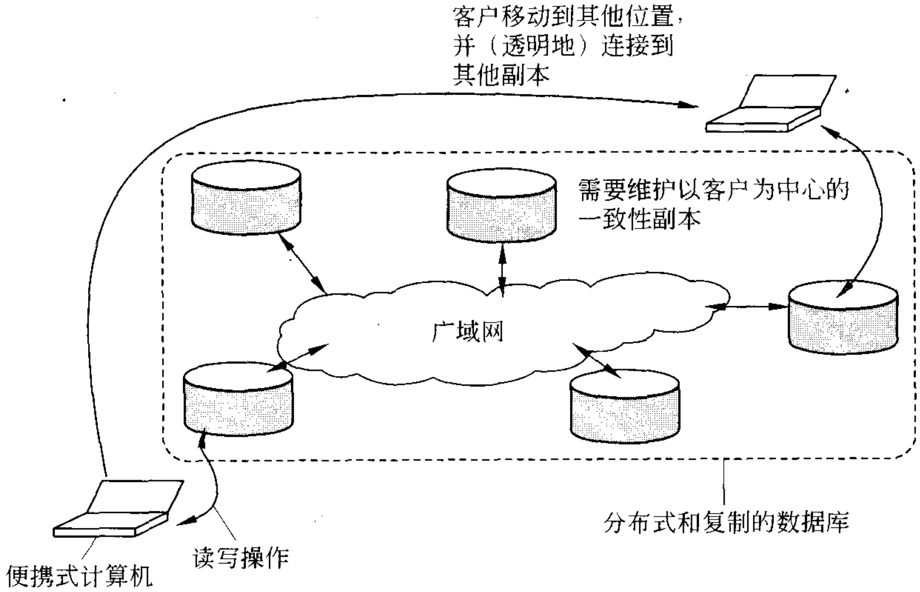


图7.11 移动用户访问分布式数据库的不同副本的原理

移动用户是以透明的方式连接到一个数据库副本来访问数据库的，也就是说，运行在用户便携式计算机上的应用程序不知道它实际在哪个副本上进行操作。假设用户执行几个更新操作，然后断开了与数据库的连接。后来，他要再次访问数据库时，他可能移动到了其他位置，或者他可能使用不同的接入设备访问数据库。此时，用户可能连接到一个与上次连接不同的副本，如图7.11所示。然而，如果先前执行的更新操作还没有传播到这个副本，那么用户就会注意到这个不一致。特别是，他可能期望看到所有先前所做的改变，但是相反，他所看到的可能是好像根本没有做过任何改变一样。

对于最终一致性的数据存储而言，这个示例很有代表性。问题是由用户有时可能对不同的副本进行操作的事实引起的，引入以**客户为中心的一致性**(client-centric consistency)可以缓解这一问题。本质上，以客户为中心的一致性为单一的客户提供一致性保证，保证该客户对数据存储的访问的一致性。它并不为不同客户的并发访问提供任何一致性保证。

以客户为中心的一致性模型起源于关于Bayou的工作(请参见(Terry 等 1994，和Terry 等 1998))。Bayou是为移动计算开发的数据库系统，该系统假设网络连接是不可靠的，而且易于受到多种性能问题的影响。无线网络和跨越范围较广的网络，例如因特网，都属于这类网络。

Bayou 本质上区分了4 种不同类型的一致性模型。为了解释这些模型，我们将再次讨论物理上分布于多台机器上的数据存储。当一个进程访问这个数据存储时，它通常连接到其本地（或最近）可用的副本上，尽管原则上任何副本都可以很好地为其提供数据。所有读、写操作都是对这个本地副本进行操作。数据更新最终将被传播到其他副本。为了简化问题，我们假设数据项拥有一个关联的所有者，这个所有者就是那个唯一被允许修改该数据项的进程。通过这种方法，我们可以避免写写操作冲突。

我们使用以下的标记法描述以客户为中心的一致性模型。设 $x_i[t]$表示 t 时刻本地副本 L_i中数据项 x 的版本。初始化后，对 L_i中的 x 进行一系列写操作将得到版本 $x_i[t]$。我们将这些操作的集合记为 $WS(x_i[t])$。如果在随后的 t_2时刻，$WS(x_i[t_1])$中的操作也已经在本地副本 L_j上执行完毕，那么我们记为 $WS(x_i[t_1];\ x_j[t_2])$。如果从上下文环境可以清楚地判断出操作顺序或时间顺序，那么时间下标将被省略。

7.3.2 单调读

第一个以客户为中心的一致性模型是单调读的一致性模型。如果数据存储满足以下条件，那么称该数据存储提供**单调读一致性**（monotonic-read consistency）：

如果一个进程读取数据项 x 的值，那么该进程对 x 执行的任何后续读操作将总是得到第一次读取的那个值或更新的值。

也就是说，单调读一致性保证，如果一个进程已经在 t 时刻看到 x 的值，那么以后它不再会看到较老版本的 x 的值。

下面，我们讨论一个分布式的电子邮件数据库，作为应用单调读一致性的一个示例。在这个数据库中，每个用户的邮箱可能分布式复制于多台机器上。邮件可以被插入到在任何位置的邮箱中。但是，数据更新是以一种懒惰（即，按需更新）方式传播的。假设一个用户在旧金山读取他的邮件。假定只读取邮件不会影响邮箱，也就是说，消息不会被删除，不会被存储到子目录下，甚至不会被标记为已读。当用户飞到纽约后，再次打开他的邮箱时，单调读一致性可以保证当他在纽约打开他的邮箱时，邮箱中仍然有旧金山的邮箱里的那些消息。

图 7.12 以图形的方式描述了单调读一致性，其中的标记法与以数据为中心的一致性模型所用的标记法相似。在垂直轴上，L_1和 L_2表示数据存储的两个不同的本地副本。水平轴表示时间。

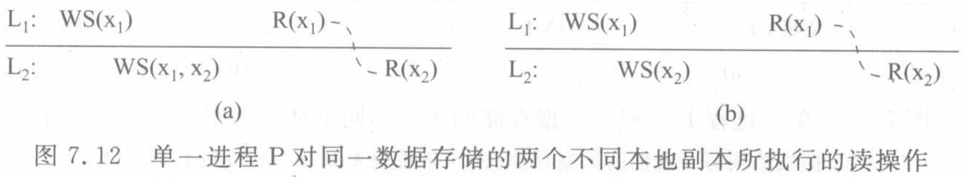


图 7.12 单一进程 P 对同一数据存储的两个不同本地副本所执行的读操作

(a) 提供单调读一致性的数据存储；(b) 不提供单调读一致性的数据存储

在图 7.12 中，进程 P 先在 L1 对 x 执行了一次读操作，得到值 X_1（在那一时刻）。该值是在 L_1执行的 $WS(x_1)$中的写操作的结果。后来，P 在 L_2对 x 执行了一次读操作，即图中的 $R(x_2)$。为了保证单调读一致性，$WS(x_1)$中的所有操作都应该在执行第二个读操作前传播到 L_2。也就是说，我们需要确保 $WS(x_1)$是 $WS(x_2)$的一部分，记为 $WS(x_1;\ x_2)$。

与之相对照，图 7.12(b)表示不保证单调读一致性的情况。进程 P 在 L_1 读取 x_1 后，又在 L_2 执行 $R(x_2)$ 操作。然而，L_2 只执行了 $WS(x_2)$ 中的写操作。此时，不能保证集合 $WS(x_2)$ 同样包含 $WS(x_1)$ 中的所有操作。

7.3.3 单调写

在很多情况下，写操作以正确的顺序传播到数据存储的所有副本是非常重要的。这种性质被描述为单调写一致性。**单调写一致性**(monotonic-write consistency)的数据存储应满足以下条件：

一个进程对数据项 x 执行的写操作必须在该进程对 x 执行任何后续写操作之前完成。

因而，完成一个写操作意味着不管后续操作的启动位置，执行这个后续操作的副本都能反映出同一进程先前执行的写操作的结果。也就是说，在数据项 x 的副本上执行的写操作只有在该副本已经通过任何先前的写操作进行更新之后才能被执行，而这些先前执行的写操作可能发生在 x 的其他副本上。

注意，单调写一致性与以数据为中心的 FIFO 一致性相似。FIFO 一致性的本质是，同一进程执行的写操作必须在任何地方以正确的顺序执行。这一顺序限制也适用于单调写一致性，只是我们这里考虑的是仅为单一进程维持的一致性，而不是为许多并发进程维持的一致性。

每个写操作完全覆盖 x 的当前值时，没有必要将 x 的副本更新为最新值。然而，写操作通常只修改数据项的部分状态。让我们来考虑一个软件库。在很多情况下，更新这种库是通过替换一个或多个函数来实现下一版本的。单调写一致性保证，如果在库的一个副本上执行数据更新，那么前面(在其他副本上)执行的所有数据更新都将首先执行。此后，所得的库将确实是最新版本的库，并将包括所有以前各个版本的更新。

图 7.13 是单调写一致性的例子。图 7.13(a)中，进程 P 在本地副本 L_1 对 x 执行一次写操作，记为操作 $W(x_1)$。然后，进程 P 对 x 再执行一次写操作，但是这次是在 L_2 处执行，如图中的 $W(x_2)$。为了保证单调写一致性，先前在 L_1 执行的写操作必须已经传播到 L_2。这就解释了图中 L_2 处出现操作 $W(x_1)$，以及它出现在 $W(x_2)$ 之前的原因。

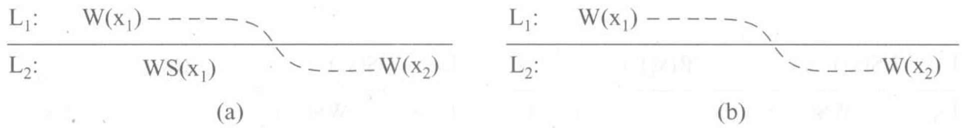


图 7.13 单一进程 P 对同一数据存储的两个不同本地副本所执行的写操作
(a) 提供单调写一致性的数据存储；(b) 不提供单调写一致性的数据存储

与之相对照，图 7.13(b)表示不保证单调写一致性的情况。与图 7.13(a)相比，图中没有向副本 L_2 传播的 $W(x_1)$。也就是说，它不能保证在 x 的副本上执行第二次写操作时，x 的副本的值与执行完 $W(x_1)$ 时的值相同或比之更新。

注意，在单调写一致性的定义中，同一进程的写操作的执行顺序与这些操作的启动顺序相同。一个相对较弱的单调写形式是写操作的结果仅当所有先前的写操作也都已经执行完

毕后才会被看到，但是它们的执行顺序可能与这些操作的最初启动顺序不同。这种一致性可应用于写操作是可交换的情况，此时，写操作的顺序的确是不必要的。详细信息请参见(Terry 等 1994)。

7.3.4 读写一致性

下面介绍一种与单调写一致性有密切关系的以客户为中心的一致性模型。如果数据存储满足以下条件，那么称该数据存储提供**读写一致性**(read-your-writes consistency)：

> 一个进程对数据项 x 执行一次写操作的结果总是会被该进程对 x 执行的后续读操作看见。

也就是说，一个写操作总是在同一进程执行的后续读操作之前完成，而不管这个后续的读操作发生在什么位置。

在更新 Web 的 HTML 页面后浏览更新效果时，经常遇到缺乏写读一致性的情况。更新操作通常是通过标准的编辑器或文字处理器实现的，它们将新版本页面存储在 Web 服务器所共享的文件系统上。用户的 Web 浏览器可能在它从本地 Web 服务器请求这个文件之后，再访问这个文件。但是，一旦文件已经被访问过，服务器或浏览器通常都会为以后的访问缓存一个本地副本。因此，当 Web 页面被更新时，如果浏览器或服务器返回高速缓存的副本而不返回原文件的话，用户将看不到更新的结果。写读一致性可以保证，如果编辑器和浏览器被集成到一个单一的程序中，那么页面更新后，高速缓存是无效的，所以将获取和显示被更新的文件。

更新密码时也会出现相同的情况。例如，Web 上的电子图书馆通常要求输入账号和相应的密码。但是，密码的修改可能需要一些时间才能生效，这可能导致用户有几分钟的时间不能进入图书馆。这种延迟可能是由于系统使用一个单独的服务器管理密码，而加密的密码传播到图书馆的多个服务器可能需要一定时间造成的。

图 7.14(a)表示了一个提供写读一致性的数据存储。注意，图 7.14(a)与图 7.12(a)非常相像，只是这里的一致性是通过进程 P 执行的最后一次写操作确定的，而不是通过进程 P 的最后一次读操作确定的。

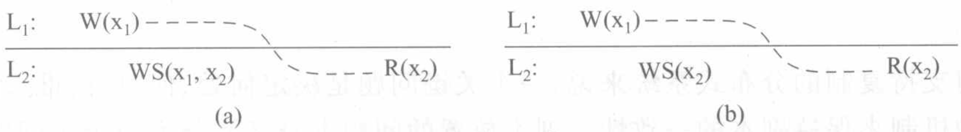


图 7.14

(a) 提供写读一致性的数据存储；(b) 不提供写读一致性的数据存储

在图 7.14(a)中，进程 P 执行了一次写操作 $W(x_1)$，然后在一个不同的本地副本处执行了一次读操作。写读一致性保证，写操作的结果可以被所有后续的读操作看到。这一过程表示为 $WS(x_1;x_2)$，它表示 $W(x_1)$是 $WS(x_2)$的一部分。与之相对照，在图 7.14(b)中，$W(x_1)$不在 $WS(x_2)$之中，这意味着进程 P 执行的前一个写操作的结果还没有传播到。

7.3.5 写读一致性

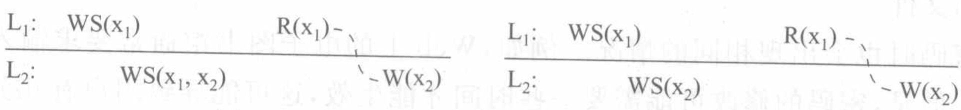
最后一种以客户为中心的一致性模型是这样的模型，即更新是作为前一个读操作的结果传播的。如果数据存储满足以下条件，那么称该数据存储提供**写读一致性**(writes-follow-reads consistency)。

> 同一个进程对数据项 x 执行的读操作之后的写操作，保证发生在与 x 读取值相同或比之更新的值上。

也就是说，进程对数据项 x 所执行的任何后续的写操作都会在 x 的副本上执行，而该副本是用该进程最近读取的值更新的。

写读一致性可以用于保证，网络新闻组的用户只有在已经看到原文章之后才能看到它的回应文章(Terry 等 1994)。为了理解这个问题，我们假设一个用户先读了一篇文章 A。然后，他回应了一篇文章 B。通过要求保证读写一致性，B 被写入新闻组的任何副本之前，A 也必须已经被写入那个副本。注意，只读取文章的用户不需要任何特定的以客户为中心的一致性模型。读写一致性保证，当且仅当原文章存储在某个本地副本上时，该文章的回应文章才被存储在这个本地副本上。

图 7.15 表示了这种一致性。在图 7.15(a)中，一个进程在本地副本 L_1 处读取 x。写入刚才所读取值的写操作也出现在 L_2 处的写操作集合中。稍后，同一进程在 L_2 处执行了一次写操作。注意，L_2 处的其他进程也看到了那些写操作。而图 7.15(b)中的数据存储无法保证 L_2 处执行的操作是在与 L_1 处所读的那个副本一致的副本上执行的。



(a) (b)

图 7.15

(a) 提供读写一致性的数据存储；(b) 不提供读写一致性的数据存储

7.4 复制管理

对任何支持复制的分布式系统来说，一个关键问题是决定何处、何时、由谁来防止副本，以及用何种机制来保持副本的一致性。副本放置的问题本身又要分为两个子问题：副本服务器放置问题和内容放置问题。这两者之间的差别很微妙但很重要，且这两个问题的界线往往不是很清楚。副本服务器的放置关心的是，找到放置托管(部分)数据存储的服务器的最佳位置。内容放置负责找到放置内容的最佳服务器。注意，这往往意味着，我们所寻找的是单个数据项的最佳位置。显然，在内容放置发生之前，必须先放置副本服务器。接下来将介绍这两种不同的放置问题，然后讨论管理被复制内容的基本机制。

7.4.1 副本服务器的放置

副本服务器的放置问题之所以没有被集中地研究，其原因很简单，它往往更多的是管理和商业问题而不是优化问题。然而，对客户和网络属性的分析有助于做出决策。

计算副本服务器的最佳位置的方法有很多，但都要归结到优化问题，其中，需要从 N 个位置中选出 K 个来(K<N)。这些问题是计算复杂性的，只能通过探索式求解。(Qiu 等 2001)以客户与位置之间的距离作为起点。距离可以通过延时或带宽来度量。假定已放置了 k 个服务器(即还剩 N-k 个位置)，该解决方法在某个时间里选取一个服务器，使得服务器与其客户之间的平均距离最小。

(Radoslavov 等 2001)提出的另一种方法是忽略客户的位置，只采用由自治系统形成的因特网拓扑。把**自治系统**(autonomous system，AS)看作为一个网络是最好不过了，其中，所有结点都运行相同的路由协议，该协议由单个组织来管理。到 2006 年 1 月为止才 20 000 个自治系统。Radoslavov 等首次考虑了最大的自治系统，在含有最大数量的网络接口(即链接)的路由器上放置一台服务器。然后，对第二大的自治系统反复使用该算法，以此类推。

事实证明，假定客户均匀地分布在因特网中(与现有的拓扑结构有关)，对客户不敏感的服务器的放置可以获得与对客户敏感的服务器的放置类似的结果。这种假设在多大程度上为真的并不清楚。人们仍没有很好地研究它。

这些算法的一个问题是，它们的计算代价高。例如，前面两种算法的复杂度都高于 $O(N^2)$，其中 N 为要查看的位置点。在实践中，这意味着，即使是几千个位置点，一个计算可能就需要运行几十分钟。这是不可接受的，特别是在出现**瞬时拥塞**(flash crowds)(即对某个站点有一个突然的高峰请求，这在因特网中是经常发生的)时。在这种情况下，快速地确定副本服务器的位置很重要，然后就可以为内容放置选择一个特定服务器。

(Szymaniak 等 2006)开发了一种方法，利用这种方法，可以快速地标识放置副本的一个区域。一个区域被标识为访问相同内容的结点集，且结点之间的延时小。这种算法的目标是首先选择最能满足需要的区域，即具有最多结点的，然后让该区域中的某个结点作为副本服务器。

最后，设想在 m 维空间中定位结点，正如前一章所介绍的那样。其基本思想是，标识 K 个最大集群，把每个集群中的结点赋给托管已复制内容。要标识这些集群，可以把整个空间划分为多个单元。选择 K 个密度最大的单元放置副本服务器。一个单元就是一个 m 维的超立方体。对二维空间来说，它对应的是一个长方形。

显然，单元的大小很重要，如图 7.16 所示。如果单元选择过大，那么会有多个结点的集群包含在同一个单元中。在这种情况下，为这些集群选择的副本服务器就会太少了。另一方面，单元过多，又会导致选择了太多的副本服务器。

事实证明，合适的单元大小可以以一个简单的函数形式来计算，该函数计算两结点之间的距离，以及所需副本的数量。可以证明，有了单元的大小，该算法的性能与近似最佳算法的同样好(Qiu 等 2001)，而且复杂性要低得多，为 $O(N\times\max\{\log(N), K\})$。实践表明，为 64 000 个结点计算 20 个最佳副本放置点大约需要 50 000 次。因此，副本服务器的放置可以实时完成。

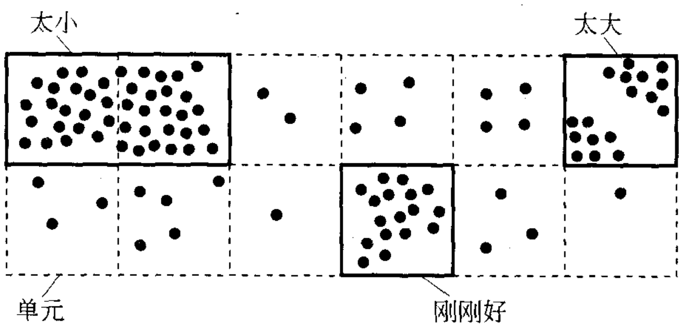


图 7.16　为服务器的放置点选择一个合适的单元大小

7.4.2　内容复制与放置

让我们把视线从服务器放置转移到内容放置上来。有关内容复制与放置，从逻辑上可以组织为三种不同类型的副本，如图 7.17 所示。

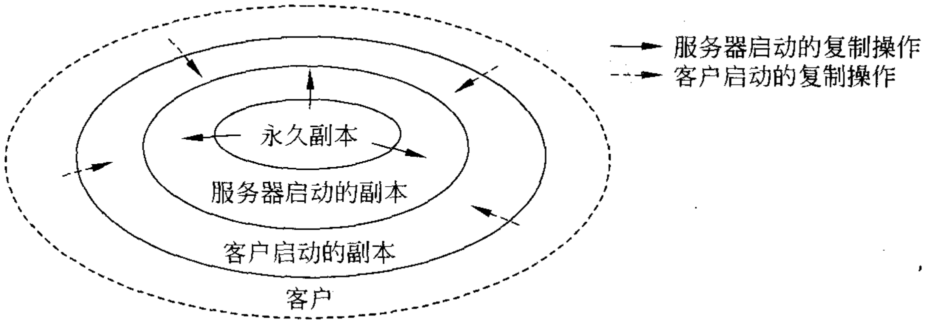


图 7.17　将数据存储的不同类型的副本逻辑地组织成三个同心环

7.4.2.1　永久副本

永久副本可以被视为构成分布式数据存储的副本的初始集。很多情况下，永久副本的数量很小。以 Web 站点为例。一个 Web 站点的分布通常采用以下两种形式之一。第一种形式的分布是，在单个位置的有限数量的服务器上复制构成站点的文件。一旦接收到请求，就将该请求转发到其中一台服务器，例如，使用循环提示器策略来进行转发。

分布式 Web 站点的第二种形式是**镜像**（mirroring）。在这种情况下，Web 站点被复制到有限数量的服务器上，这些服务器称为**镜像站点**（mirror site），它们在地理上散布于因特网之中。在很多情况下，客户只是从提供给他们的镜像站点列表中选择一个镜像站点。被镜像的站点与基于群集的 Web 站点有共同之处，即站点只有少量的副本，而这些副本或多或少是使用静态配置的。

相似的静态组织方式也出现在分布式数据库中（Oszu 和 Valduriez 1999）。同样，数据库可以被分布地复制在很多服务器上，这些服务器一起形成服务器群，通常称之为**无共享体系结构**（shared-nothing architecture），该体系结构强调处理器既不共享磁盘也不共享内存。另外，数据库也可能分布地被复制在地理上分散的几台服务器上。在联合数据库（Sheth 和 Larson 1990）中通常部署这种体系结构。

7.4.2.2 服务器启动的副本

与永久副本相对，服务器启动的副本是为提高性能而存在的数据存储的副本，该副本是在初始化数据存储的所有者时创建的。例如，位于纽约的一个 Web 服务器就是这样的。一般，这台服务器可以相当方便地处理输入的请求，但是可能发生以下情况：它突然在几天内接收到来自远离服务器的意外位置的大量请求（最近的 Web 历史中已经在一些场合出现过这种突发大量请求情况）。在这种情况下，在产生请求的地区安装一些暂时副本可能是值得的。

Web 托管服务解决了动态放置副本的问题。这些服务在本质上提供了一个散布于因特网的（相对静态的）服务器集。这些服务器可以维护属于第三方的 Web 文件，并提供对这些文件访问。为了提供优化的功能，这些托管服务可以动态地把文件复制到需要这些文件以提高性能的服务器，也就是接近发出请求的客户的服务器那里。

服务器启动的副本的一个问题是决定创建或删除副本的确切位置和时间。(Rabinovich 等 1999)描述了一种 Web 托管服务中实现文件动态复制的方法。设计该算法的目的是为了支持 Web 页面可在哪些原因下假设更新的个数与读请求的个数相比相对较少。该算法使用文件作为数据单元，其工作方式如下所述。

动态复制的算法需要考虑两个问题。第一，复制可能是为了减轻一台服务器的负载而进行的；第二，一台服务器上的指定文件可能被转移或复制到对这些文件提出很多请求的客户附近的服务器。下面，我们只集中考虑第二个问题。我们也不考虑一些细节问题，关于这些细节问题请见(Rabinovich 等 1999)。

每台服务器跟踪每个文件的访问计数以及提出这些访问请求的位置。特别是，假设对于一个给定的客户 C，每台服务器都可以确定 Web 托管服务中的哪台服务器最靠近 C（例如，可以从路由数据库获得这一信息）。如果客户端 C_1 和客户端 C_2 共享最靠近的服务器 P，那么所有来自 C_1 和 C_2 对服务器 Q 的文件 F 的访问请求一起被 Q 记入单一的访问计数 cntQ(P,F)之中。图 7.18 显示了这种情形。

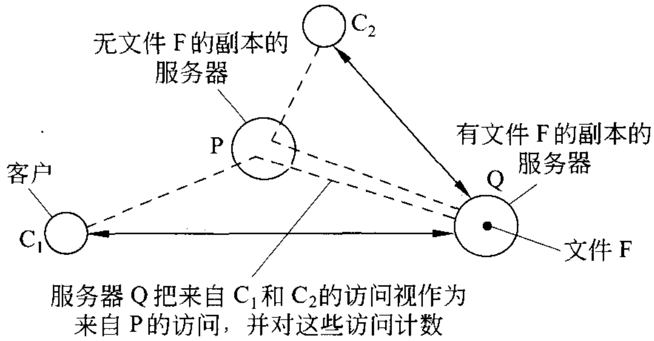


图 7.18 对来自不同客户的访问请求计数

当对服务器 S 上的指定文件 F 的请求数量下降到删除阈值 del(S, F)之下时，服务器 S 可以删除该文件。因而，该文件的副本的数目减少了，这可能导致其他服务器的工作负载增加。可采用特殊的方法来保证每个文件至少仍存在一个副本。

复制阈值 rep(S, F)指出对指定文件的请求数量过高，以至于值得将该文件复制到另

一台服务器上。通常所选择的复制阈值比删除阈值高。如果请求的数量位于删除阈值和复制阈值之间，那么只允许转移该文件。也就是说，在这种情况下，至少保持该文件的副本的个数不变，这一点是十分重要的。

当服务器 Q 决定重新评估它存储的文件的位置时，它检查每个文件的访问计数。如果服务器 Q 上文件 F 的访问请求的总数低于删除阈值 del(Q, F)，那么它将删除 F，除非 F 是最后一个副本。此外，如果对于某台服务器 P，cntQ (P, F) 超过 Q 上 F 的请求总数的半数以上，服务器 Q 请求服务器 P 接管 F 的副本。也就是说，服务器 Q 将试图把 F 转移到 P。

向服务器 P 转移文件 F 不可能总能成功。例如，因为 P 的负载总是很重或者 P 没有磁盘空间就会造成文件转移不成功。在这种情况下，Q 将试图在其他服务器上复制 F。当然，只有对 Q 上的 F 的访问请求的总数超过复制阈值 rep(Q, F) 时才会发生复制。服务器从 Web 托管服务中最远的服务器开始，检查其中所有的服务器。如果对于某台服务器 R，cntQ(R, F) 超过对 Q 上的 F 的所有请求，并达到某个比例，Q 就会试图把 F 复制到 R。

服务器启动的复制正在逐渐地流行起来，特别是在上面介绍的 Web 托管服务的环境下。注意，只要可以保证每个数据项至少由一台服务器托管，那么只使用服务器启动的复制而不使用任何永久副本就足够了。然而，永久副本仍常常用作备份工具或用作允许被修改以保证一致性的唯一副本，而服务器启动的副本则被用于在客户附近放置只读副本。

7.4.2.3 客户启动的副本

一类重要的副本是在客户初始化时创建的副本。普遍将客户启动的副本称为**客户高速缓存**(Client cache)。其实，高速缓存是一种本地存储工具，客户使用它暂时存储刚刚请求的数据的副本。在原则上，高速缓存的管理是完全由客户负责的。客户从中获取数据的数据存储不负责保持高速缓存数据的一致。然而，如同我们所看到的，在很多情况下，客户可能要依靠数据存储通知它被缓存的数据什么时候过时。

客户高速缓存只用于改善数据的访问时间。通常，当一个客户想要访问某些数据时，它连接该数据存储的最近的副本，从该副本获得它想读取的数据，或者将它刚刚修改的数据写入该副本。当大部分操作仅是读取数据时，客户在附近的高速缓存中存储所请求的数据可以提高性能。这个高速缓存可以位于客户所在的机器中，也可以位于客户所在局域网中的另一台单独的机器中。下一次需要读取该数据时，客户可以简单地从这个本地高速缓存中获得数据。只要读取所获得的数据时，数据没有被改变，这种模式会工作得很好。

例如，通常数据将在高速缓存中保存一段有限的时间，这样做是为防止使用完全过时的数据，或只是为其他数据腾出空间。当被请求的数据可以从本地副本获得时，我们称其为**高速缓存命中**(cache hit)。为了提高高速缓存命中数量，可以让多个客户共享高速缓存。其基本假设是来自客户 C1 的数据请求可能也对来自客户 C2 附近的其他客户的请求有用。

这一假设的正确性在很大程度上取决于数据存储的类型。例如，传统文件系统几乎不共享文件，参见(Muntz 和 Honeyman 1992，Blaze 1993)，这使得共享高速缓存毫无意义。同样，事实证明，使用 Web 高速缓存来共享数据已失去市场了，部分原因是由于网络和服务器性能的提高。

客户高速缓存的放置也相对简单：通常将高速缓存放置在客户所在的机器上，或者将其放置在客户所在的局域网上由各个客户共享的机器上。然而，在某些情况下，系统管理员

会引入更高级的高速缓存。他们在一些部门或组织之间放置共享高速缓存，或者甚至在整个地区，例如省或国家内放置共享高速缓存。

另一种方法是在广域网的几个特定点放置高速缓存服务器，让客户查找距离最近的服务器。当客户查找到最近的服务器后，它请求这台服务器保留它先前从其他服务器获得的数据的副本（Noble 等 1999）。我们将在本章后面讨论一致性协议时再继续讨论缓存问题。

7.4.3 内容分发

复制管理还负责（已更新）内容往相关副本服务器上的传播。这是有代价的，稍后我们将介绍。

7.4.3.1 状态与操作

一个重要的设计问题是将要实际传播哪些信息。基本上有三种可能：

(1) 只传播更新的通知；

(2) 把数据从一个副本传送到另一个副本；

(3) 把更新操作传播到其他副本。

无效化协议（invalidation protocol）使用传播通知方式。无效化协议通知其他副本已发生了更新操作，这些副本所包含的数据不再有效。无效化消息可以指定数据存储的哪些部分被更新了，那么实际上只有部分副本是无效的。重要的问题是只传播一个通知而不传播别的消息。每当请求在无效的副本上执行操作时，一般需要先根据数据存储支持的特定的一致性模型更新那个副本。

无效化协议的主要优点是它几乎不占用网络带宽。所需传送的唯一信息是关于哪些数据不再有效的说明。当更新操作与读操作相比较多时，也就是说，读写比率相对很小时，这种协议通常工作得非常好。

例如，在一个数据存储中，更新是通过向所有副本发送被修改的数据传播的。如果被修改的数据很多，并且更新操作与读操作相比发生得更频繁，那么我们可能遇到以下情况：接连发生两个更新操作，而这两个更新操作之间没有读操作。因而，第一次向所有副本传播的更新实际上是无用的，因为它将被第二个更新操作的结果覆盖。相反，在这种情况下，发送数据已被修改的通知可能更加高效。

在多个副本间传送被修改的数据是第二种选择，在读对写比率相对较高时，这种方法十分有用。在这种情况下，更新是有效的可能性较高，因为从某种意义上来说，被修改的数据会在下一个更新操作发生前被读取。不传播被修改的数据，而将这些修改记入日志，然后仅传送那些日志以节约带宽也是可以的。另外，更新经常是通过将多个修改压缩到一个消息中的方式被组合传送的，从而减少通信开销。

第三种方法根本不传送任何数据修改，而告诉每个副本它应该执行的更新操作，这种方法又称为**主动复制**（active replication）。它假设每个副本由一个进程代表，该进程能够通过执行操作来“主动地”保持它所关联的数据为最新的数据（Schneider 1990）。主动复制的主要优点是假设一个操作所关联的参数相对较少，传播更新的带宽代价通常可以达到最小。另一方面，每个副本可以获得更多的处理能力，在操作相对复杂的情况下更是如此。

7.4.3.2 拉协议与推协议

另一个设计问题是，更新是拉式的还是推式的。**基于推式的方法**（push-based approach）又称为**基于服务器的协议**（server-based protocol）。在这种方法中，甚至不需要其他副本请求更新，这些更新就被传播到那些副本那里。永久副本和服务器启动的副本之间通常使用基于推式的方法。基于服务器的协议应用于多个副本常常需要维持相对较高程度的一致性的时候，也就是说，当副本需要保持完全相同的时候。

需要保持较高程度的一致性是基于这样一个事实：通常有许多客户共享永久副本和服务器启动的副本，以及较大的共享的高速缓存，而这些客户轮流地执行操作，主要执行读操作。因而，每个副本上读与更新之比率相对较高。在这些情况下，基于推式的协议是高效的，因为每个被推的更新可以对一个或更多读程序有用。另外，基于推式的协议使一致的数据在被请求时立即有效。

相反，在**基于拉式的方法**（pull-based approach）中，一台服务器或客户请求其他服务器向它发送该服务器此时持有的任何更新。基于拉式的协议，又称为**基于客户的协议**（client-based protocol），通常被用于客户高速缓存。例如，Web高速缓存使用的一种通用策略是先检查被缓存的数据项是否仍然是最新的。当高速缓存接收到对数据项的请求，而该数据项仍对本地有效的时候，高速缓存查看原始的Web服务器，以确定那些数据项自从被缓存后是否被修改过。如果那些数据项被修改过，那么被修改的数据首先被传送到高速缓存，然后才被返回给请求数据的客户。如果那些数据项没有被修改过，那么就向客户返回被缓存的数据。也就是说，客户轮询服务器以查看是否需要更新高速缓存。

在读与更新之比率相对较低时，基于拉式的方法是高效的。非共享的客户高速缓存通常属于这种情况，此时客户高速缓存只有一个客户。然而，即使在许多客户共享一个高速缓存，在被缓存的数据项很少被共享时，也可证明基于拉式的方法是高效的。与基于推式的方法相比，基于拉式的策略的主要缺点是在高速缓存没有命中时，响应时间会增大。

比较基于推式的方法和基于拉式的方法时，需要做出一些权衡，如图7.19所示。简单起见，我们以一个客户-服务器系统为例，该系统由一个单一的、非分布式的服务器和一些客户进程构成，每个客户进程都有它们自己的高速缓存。

问题	基于推式	基于拉式
服务器的状态	客户副本和高速缓存的列表	无
发送的消息	更新（以及以后可能获取的更新）	轮询和更新
客户响应时间	立即（或获取更新的时间）	获取更新的时间

图7.19　在多客户、单一服务器的系统中，基于推式的协议和基于拉式的协议的比较

基于推式的协议中的一个重要问题是服务器需要跟踪所有的客户高速缓存。状态较多的服务器不但通常缺少容错能力（如第3章所述），而且跟踪所有的客户高速缓存也可能在服务器端引入相当大的额外开销。例如，在基于推式的方法中，Web服务器可能需要跟踪数以万计的客户高速缓存。每次Web页面被更新时，服务器将需要遍历持有该页面的副本的客户高速缓存的列表，然后向这些客户高速缓存传播更新。但是更糟的是，如果客户由于缺少空间而清除了一个页面，那么它必须通知服务器，这又导致了更多的通信。

在这两种方法中，需要在客户和服务器之间发送的消息也有所不同。在基于推式的方法中，唯一的通信是服务器向每个客户发送更新。当更新实际上仅是无效化消息时，客户需要进行其他通信以获取被修改的数据。在基于拉式的方法中，客户必须轮询服务器，并在必要时获取被修改的数据。

最后，在两种方法中客户响应时间也是不同的。当服务器将被修改的数据推入客户时，客户的响应时间显然是0。当推入无效化消息时，响应时间与基于拉式的方法中的响应时间相同，该响应时间由客户从服务器获取被修改的数据的时间决定。

这些代价权衡导致出现了一种更新传播的混合形式——基于租用的更新传播。**租用**(lease)是服务器所作的承诺，它将在指定的时间内把更新推给客户。当租用到期时，客户被迫轮询服务器以实现更新，并在必要时拉出被修改的数据。另一种方法是在前一个租用到期时，客户请求一个新的租用以实现更新的推入。

租用的概念最初是由Gray和Cheriton(1989)提出的。这是他们为基于推式的策略和基于拉式的策略之间的动态转换提供的一种便利机制。(Duvvuri等2000)描述了一种灵活的租用系统，该系统允许租用期限根据不同租用标准动态调整。他们对以下三种类型的租用进行区分(注意，在所有的情况下，只要租用没有到期，更新都是由服务器负责推入的)。

首先，是一种基于数据项"年龄"的租用，数据项的"年龄"取决于该数据项最后一次被修改后的延续时间。其基本假设是长时间内未被修改的数据可能会在将来的一段时间内仍不会被修改。基于Web的数据已表明这一假设是合理的。与所有租用具有相同过期时间的方式相比，为预期保持不变的数据项授予一个长期的租用可以使更新消息的个数大大地减少。

另一种租用是基于特定客户请求更新高速缓存副本的频率的租用。使用基于更新频率的租用时，服务器将为需要经常刷新其高速缓存的客户分配一个长期的租用，而为只偶尔请求特定数据项的客户分配一个使用该数据项的短期租用。该策略的效果是服务器基本上只跟踪其数据受欢迎的那些客户，此外，那些客户端也被提供了较高程度的一致性。

最后一种租用是基于服务器的状态空间开销的租用。当服务器意识到它会逐渐过载时，它会降低分配给客户的新租用的使用期限。这一策略的效果是服务器需要跟踪的客户变少，这是因为租用到期很快。也就是说，服务器动态地转换到一个状态较少的操作模式，从而减轻了服务器负载以使其可以更高效地处理请求。

7.4.3.3　单播与多播

与推式更新或拉式更新相关的一个问题是决定应使用单播还是多播。在单播通信中，当作为数据存储的一部分的服务器向其他N台服务器发送其更新时，它通过发送N个单独的消息实现，即向每台服务器发送一个消息。使用多播通信时，底层的网络负责向多个接收者高效地发送一个消息。

在很多情况下，使用可用的多播工具是比较廉价的。一种极端的情况是所有副本都位于同一个局域网，此时，硬件广播是可用的。在这种情况下，广播或多播一个消息的开销并不比一个单一的点对点消息传送开销大。但单播更新的效率会较低。

与基于推式的方法结合时，多播常常可以高效地传播更新。在这种情况下，一个决定向其他一些服务器推出更新的服务器仅使用一个单一的多播组来发送它的更新。相反，使用

基于拉式的方法时，通常只有一个单一的客户或服务器请求更新其副本。在这种情况下，单播可能是最高效的方法。

7.5 一致性协议

到目前为止，我们的注意力主要集中在各种一致性模型和一致性协议的一般设计问题。本节我们通过几个一致性协议的示例来集中讨论一致性模型的实际实现。**一致性协议**(consistency protocol)描述了特定一致性模型的实现。我们按照有关一致性模型的组织结构，先介绍以数据为中心的模型，然后介绍以客户为中心的模型。

7.5.1 持续一致性

作为关于持续一致性工作的一部分，Yu 和 Vahdat 还开发了一些处理三种一致性的协议。下面我们将简要介绍一些解决方法，为清楚起见，省去了一些细节内容。

7.5.1.1 限定复制的数字偏差

我们首先来看一种把数值偏差限定在一定范围内的解决方法。这里的目的同样不是深入每个协议的细节，而是给出一般的思想。有关限定数值偏差的详细内容请参见(Yu 和 Vahdat 2000b)。

我们来看看往单个数据项 x 写数据的操作。每个写操作 W(x)都有一个相关的权重，表示一个用于更新 x 的数值，记作 weight(W(x))，或简写为 weight(W)。为简单起见，我们假设 weight(W)>0。开始时，每个写操作 W 被提交给 N 个可用副本服务器之中的一个服务器，在这种情况下，该服务器就成为了该写操作的源，记作 origin(W)。当我们在某个特定的时间考察系统时，可以看到会有多个已提交的写操作需要传播给所有服务器。最后，每个服务器 S_i 会保持一个写操作日志 L_i，该日志记录了服务器在其本地副本 x 上执行的写操作。

设如下的 TW[i, j]是源自服务器 S_j 并由服务器 S_i 运行的写操作：

$$TW[i,j] = \sum \{weight(W) \mid origin(W) = S_j \,\&\, W \in L_i\}$$

注意，TW[i, j]表示的是提交给 S_i 的总的写操作。我们的目标是，对任意时间 t，使得服务器 S_i 的当前值 v_i 在数据项 x 的实际值 v(t)的一定范围内发生偏差。该实际值完全由所有已提交的写操作来确定。即，如果 v(0)是 x 的初始值，那么

$$v(t) = v(0) + \sum_{k=1}^{N} TW[k,k]$$

$$v_i \doteq v(0) + \sum_{k=1}^{N} TW[i,k]$$

注意，其中 $v_i \leqslant v(t)$。这里我们只关注绝对偏差。特别地，对每个服务器 S_i，指定一个上限 δ_i，这样就需要确保

$$v(t) - v_i \leqslant \delta_i$$

提交给服务器 S_i 的写操作需要传播给其他所有服务器。这里有不同的方法来完成，其中，感染协议可以实现更新的快速传播。在任何情况下，当服务器 S_j 把来自 S_i 的一个写操作传播给 S_k 时，S_k 可以知道写操作发送时的值 TW[i, j]。换句话说，S_k 可以维护一个视图 $TW_k[i, j]$，它相信 S_j 会有一个 TW[i, j]值。显然，

$$0 \leqslant TW_k[i, j] \leqslant TW[i, j] \leqslant TW[j, j]$$

整个思想是，当服务器 S_k 知道 S_i 与提交给 S_k 的更新操作步调不一致时，它就把写操作从其日志中转发给 S_i。该转发操作可以有效地把 S_k 的视图 $TW_k[i, k]$ 往 TW[i, k]靠近，使得偏差(TW[i, k]－$TW_k[i, k]$)更小。尤其是当应用程序提交一个新的写操作时，S_k 会把其视图往 TW[i, k]推进，这将使得(TW[i, k]－$TW_k[i, k]$)大于 $\delta_i/(N-1)$。本章后面有一个练习，证明这种推进可以确保 $v(t)-v_i \leqslant \delta_i$。

7.5.1.2　限定复制的新旧程度偏差

有很多种方法把副本的新旧程度限定在指定的范围内。一种简单的方法让服务器 S_k 保持一个实时的向量时钟 RVC_k，其中，$RVC_k[i]=T(i)$ 意为到时间 T(i)时，S_k 看到了已提交给 S_i 的所有写操作。在这种情况下，我们假定每个已提交的写操作由其初始服务器添加时间戳，T(i)表示的是 S_i 的本地时间。

如果副本服务器之间的时钟是松散同步化的，那么一个可接受的限定新旧程度的协议是这样的：只要服务器 S_k 通知 T[k]－$RVC_k[i]$ 将超出指定的界限，那么它就开始拉入来自 S_i 的时间戳晚于 $RVC_k[i]$ 的写操作。

注意，在这种情况下，副本服务器按照在其他地方发出的写操作，负责使其 x 的副本保持为最新的。相反，当要维护数值界限时，使用的是一个推方法，通过转发写操作来让原始服务器使副本保存为最新的。在限定新旧程度的情况下，推写操作的问题是，当事先不知道最大的传播时间时，无法保证一致性。这种情况可以通过拉入更新得到改善，因为多个服务器有助于使服务器的 x 的副本保持为最新的。

7.5.1.3　限定顺序偏差

在持续一致性中，顺序偏差是由以下事实引起的：副本服务器会暂时性地应用已提交给它的更新。因此，每个服务器都有一个暂时写操作的本地队列，这些写操作应用到 x 的本地副本中的顺序还需确定。通过指定暂时写操作队列的最大长度，就可以限定顺序偏差。

因此，检测何时需要执行顺序一致性很简单：当本地队列的长度超过指定的最大长度时。此时，服务器不再接受任何新提交的写操作，而是按照应执行的写操作的顺序，通过与其他服务器协商，尝试提交暂时写操作。换句话说，就是需要执行一个暂时写操作的全局一致的顺序。要这样做有很多种方法，但在实践中使用较多的是基于主备份的协议和基于法定数量的协议。下面我们来讨论这些协议。

7.5.2　基于主备份的协议

在实践中，我们可以看到，分布式应用程序所遵循的一致性模型往往是相当容易理解的。这些模型包含有用于限定新旧偏差的模型，以及用于限定数值偏差的模型。关于处理

操作的一致顺序的模型，顺序一致性，尤其是其操作可以通过加锁或事务来分组的模型很流行。

对应用程序开发人员来说，一致性模型一旦变得稍微有些难以理解，即使性能可以提高，它们也会被忽视。这里的底线是，如果一致性模型的语义直观上不清楚，应用程序开发人员要构建正确的应用程序就很困难。简单性是值得肯定的(合理性大概也是如此)。

事实证明，在顺序一致性中，基于主备份的协议比较盛行。在这些协议中，数据存储中的每个数据项有一个相关的主备份，该主备份负责协调在 x 上的写操作。根据主备份是否固定在一个远程服务器上，还是将主备份移动到启动写操作的进程那里之后写操作是否可在本地执行，可以区分各种基于主备份的协议。下面我们来看看这类协议。

7.5.2.1 远程写协议

支持复制的基于主备份的最简单协议是一种所有读操作和写操作都转发给单个固定的远程服务器的协议。读操作可以在本地执行。这种方案又称为**主备份协议**(primary-backup protocol)(Budhiraja 等 1993)。主备份协议的工作原理如图 7.20 所示。要在数据项 x 上执行一个写操作的进程，会把该操作转发给 x 的主服务器。该主服务器在其 x 的本地副本上执行更新操作，随后把该更新转发给备份服务器。每个备份服务器也执行这个更新操作，并往主服务器回送一个确认消息。当所有备份服务器都更新了它们的本地副本后，主服务器回送一个确认消息给初始进程。

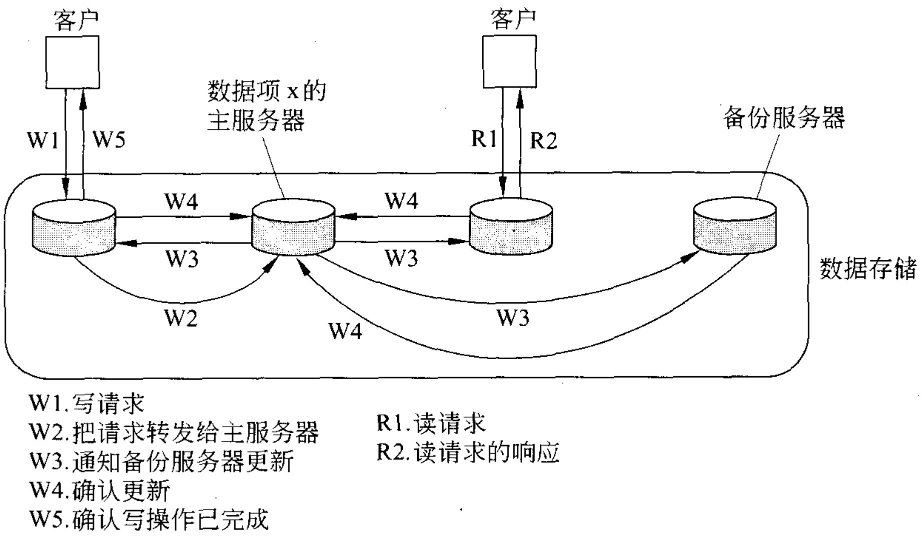


图 7.20 主备份协议的原理

这种方法存在一个潜在的性能问题，就是启动更新的进程在被允许继续执行前，可能需要等待相对较长的时间。实际上，在这种方法中，更新是以一种阻塞操作的方法来实现的。另一种方法是使用非阻塞的方法。只要主服务器已经更新了其 x 的本地副本，它就返回一个确认消息。然后，它再通知备份服务器也执行这个更新。有关非阻塞的主备份协议参见(Budhiraja 和 Marzullo 1992)。

非阻塞的主备份协议中的主要问题是它的容错能力。在阻塞方法中，客户端进程确切地知道其他若干台服务器备份了更新操作。但是在非阻塞的方法中，情况并不是这样。我

们将在下一章展开讨论容错问题。

主备份协议提供了一种实现顺序一致性的简单方法，因为主服务器可以对所有进来的写操作进行全局时间排序。显然，无论使用哪个备份服务器执行读操作，所有进程都会以相同的顺序看到所有的写操作。同样，在阻塞协议中，进程将总会看到它们最近执行的写操作的结果（注意，非阻塞协议不采取特殊措施时不能保证这一点）。

7.5.2.2 本地写协议

主备份协议的一种变体是，其主副本在要执行写操作的进程之间迁移。如前所述，当某个进程要更新数据项 x 时，它先定位 x 的主副本，然后把它移到自己的位置上，如图 7.21 所示。这种方法的主要优点是，可以在本地执行多个连续的写操作，而读操作仍可以访问其本地副本。但是，只有使用非阻塞协议，在主备份完成更新后，通过该协议将更新传播到其他副本时，才可能实现这个优点。

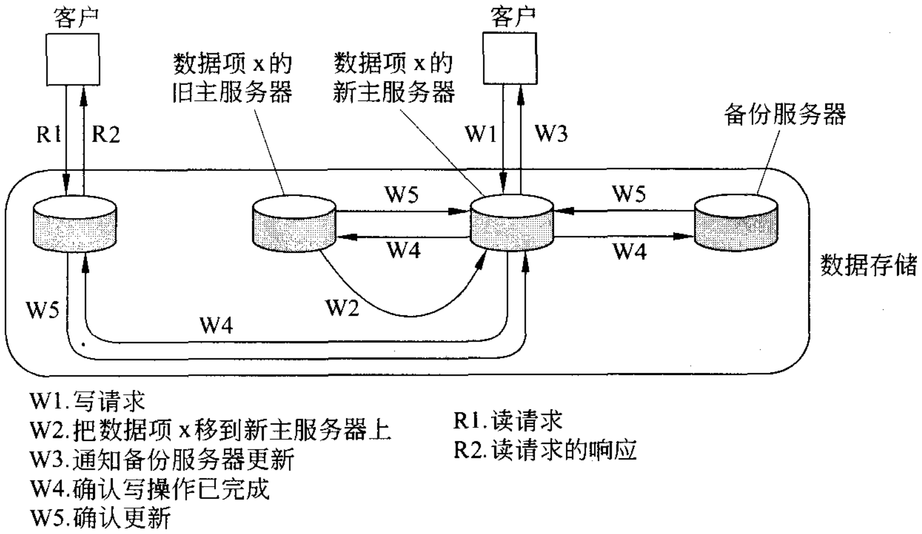


图 7.21 主备份协议，其中主备份移动到要执行更新的进程那里

主备份的本地写协议也可应用于能够在离线模式下操作的移动计算机。离线前，移动计算机成为每个它期望更新的数据项的主服务器。离线时，所有更新操作都在本地执行，而其他进程仍可以执行读操作（但不能执行更新操作）。稍后，再次连接时，更新从主服务器传播到备份服务器，使数据存储再次达到一致的状态。我们将在第 11 章讨论分布式文件系统时，再来讨论离线方式的操作。

非阻塞、本地写的基于主机的协议是这种方案的最后一个变体，它常常用于分布式系统。在这种情况下，有一个固定的中心服务器，通过它，可以进行所有的写操作，就像是远程写主备份一样。但是，该服务器可以临时允许一个副本服务器执行一系列的本地更新操作，因此可以显著地提高性能。当副本服务器完成后，更新操作被传播给中心服务器，从这里再把它们分发给其他的副本服务器。

7.5.3 复制的写协议

在复制的写协议中，写操作可以在多个副本上执行，而不是像基于主备份的副本那样只

在一个副本上执行。主动复制和基于多数表决的一致性协议的区别在于：主动复制中的操作被转发到所有副本。

7.5.3.1 主动复制

在主动复制中，每个副本有一个相关联的进程，该进程执行更新操作。与其他协议相比，更新通常是通过引起更新的写操作来传播的。也就是说，该操作被发送到每个副本。但是，使用前面讨论的方法发送更新也是可以的。

主动复制的一个问题是操作需要在各地以相同的顺序执行，因而需要一种全序的多播机制。这种多播可以用前一章所讨论的 Lamport 时间戳来实现。但是，Lamport 时间戳用于大型分布式系统时的扩展性不好。一种替代的方法是使用中心协调器，又称为**定序器**(sequencer)来实现全序。一种方法是先将所有操作转发到定序器，由定序器为每个操作分配一个唯一的序列号，随后将这些操作转发给所有副本。操作按照其序列号顺序执行。显然，这种全序多播的实现与基于主备份的一致性协议非常相似。

注意，使用定序器并不能解决可扩展性问题。事实上，如果需要全序的多播机制，那么结合 Lamport 时间戳和定序器的对称多播可能是必不可少的。这种解决方法在(Rodrigues 等 1996)中有所描述。

7.5.3.2 基于多数表决的协议

支持复制的写协议的另一种方法是使用**表决**(voting)，该方法最先由 Thomas(1979)提出，并由 Gifford(1979)一般化。其基本思想是要求客户在读或写一个复制的数据项之前向多个服务器提出请求，并获得它们的许可。

下面以一个简单的示例来说明该算法工作原理。考虑一个分布式文件系统，假设其文件被复制在 N 个服务器上。我们规定：要更新一个文件，客户必须先联系至少半数加一个服务器(即多数服务器)，并得到它们同意后执行更新。一旦它们同意更新，该文件将被修改，这个新文件也将与一个新版本号关联。该版本号用于识别文件的版本，对于所有新更新的文件，它们的版本号是相同的。

要读取一个复制文件，客户也必须联系至少半数加一个服务器，请求它们返回该文件关联的版本号。如果所有的版本号一致，那么该版本必定是最新的版本，这是因为剩余服务器的个数不够半数以上，试图只更新剩余服务器的请求将会失败。

例如，如果系统有 5 个服务器，一个客户确定其中 3 个服务器持有第 8 版本的文件，那么其余 2 个服务器不可能持有第 9 版本的文件。毕竟，所有成功地从第 8 版本到第 9 版本的更新需要得到 3 个而不只是 2 个服务器的许可。

实际上，Gifford 的方案比这个方法更加通用。在 Gifford 的方案中，一个客户要读取一个具有 N 个副本的文件，它必须组织一个**读团体**(read quorum)，该读团体是 N_R 个以上服务器的任意集合。同样地，要修改一个文件，客户必须组织一个至少有 N_W 个服务器的**写团体**(write quorum)。N_R 和 N_W 的值应满足以下两个限制条件：

(1) $N_R + N_W > N$；

(2) $N_W > N/2$。

第一个限制条件用于防止读写操作冲突，而第二个限制条件用于防止读读操作冲突。

只有在适当个数的服务器同意参与文件的读写后,客户才能读或写该文件。

为了理解该算法的工作方式,我们以图 7.22(a)为例,其中 $N_R=3$ 且 $N_W=10$。假设最近的写团体由服务器 C~L 的 10 个服务器组成。任何随后由三个服务器组成的读团体必须至少包含一个该集合中的服务器。客户查看版本号时,它会知道哪个服务器是最新的,并选择那个服务器。

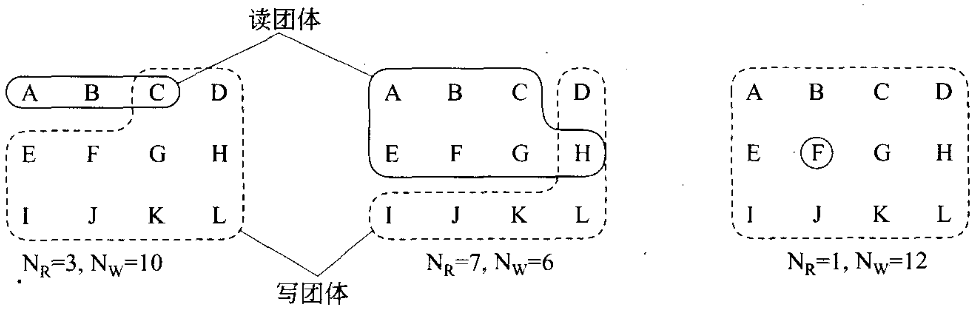


图 7.22 表决算法的三个示例

(a) 读操作集和写操作集的正确选择;(b) 可能导致写写操作冲突的读操作集和写操作集;

(c) 读操作集和写操作集的正确选择,称为 ROWA

在图 7.22(b)和(c)中,我们看到了另外两个示例。图 722(b)中的示例可能发生写写操作冲突,这是因为 $N_W \leqslant N/2$。特别地,如果一个客户选择{A,B,C,E,F,G}作为它的写操作集,而另一个客户选择{D,H,I,J,K,L}作为它的写操作集,那么显然会遇到麻烦,因为我们接受了这两个更新,没有检测是否会发生冲突。

图 7.22(c)所示的情形特别有趣,因为它设置 N_R 为 1,这使它可以通过找到并使用复制的文件的任何副本来读取该文件。但是,它所付出的代价是写更新需要获取所有副本。这种方法通常称为**“读一个,写全部”**(read-one,write-all,ROWA)。基于多数表决的复制协议有多种变形。(Jalote 1994)很好地总结了这些协议。

7.5.4 高速缓存相关性协议

高速缓存形成了一类特殊的复制,因为它们通常由客户而不是服务器控制。高速缓存相关性协议保证高速缓存与服务器启动的副本一致,在原理上,高速缓存相关性与迄今为止我们讨论的各种一致性协议有没有多大差别。

关于高速缓存的设计与实现方面的研究很多,特别是在共享存储器的多处理器系统环境中。很多方法基于底层硬件的支持,例如假设可以实现侦听或高效的广播。基于中间件的分布式系统建立在通用的操作系统之上,在这种环境中,基于软件来解决高速缓存问题的方法备受关注。此时,通常使用两个单独的标准来划分高速缓存协议(Min 和 Baer 1992,Lilja 1993,Tartalja 和 Milutinovic 1997)。

首先,不同的高速缓存解决方法在**相关性检测策略**(coherence detection strategy)上有区别,即实际检测不一致性的时间不同。静态的解决方法假定编译器在运行之前执行必要的分析,并确定哪些数据可能因为它们被缓存而实际导致了不一致性。编译器仅插入一些避免不一致性的指令。动态解决方法通常应用于本书所研究的各种分布式系统。在这些解

决方法中，不一致性是在运行时检测的。例如，检查服务器，查看被缓存的数据自从被缓存后是否被改变过。

在分布式数据库中，根据事务处理期间进行检测的确切时间，可将动态的基于检测的协议进一步精确分类。Franklin 等(1997)将其分为以下三类。第一类，在事务处理期间访问高速缓存的数据项时，客户需要检验该数据项是否仍与(可能是复制的)服务器中的该数据项一致。在检查完一致性之前，事务处理不能开始使用缓存的数据。

第二类方法是在进行一致性检验时，继续执行事务处理，该方法是最优的方法。它假设事务处理启动时高速缓存的数据是最新的。如果稍后证实该假设是错误的，那么事务处理将不得不异常中止。

第三类方法是只有事务处理委托检验时才检验高速缓存的数据是否是最新的。这类方法与上一章所讨论的乐观并发控制方法类似。实际上，事务处理只是启动对缓存的数据的操作，并希望得到令人满意的结果。在所有工作都完成后，再检验被访问的数据的一致性。如果事务处理使用的是过时的数据，那么事务处理被异常中止。

高速缓存相关性的另一个设计问题是**相关性实施策略**(coherence enforcement strategy)，该策略决定高速缓存如何与服务器存储的各副本保持一致。最简单的解决方法是完全不允许缓存共享数据，而是将共享数据仅保存在服务器上。服务器使用前面讨论的基于主备份的协议或复制的写协议来维护一致性，客户只允许缓存私有数据。显然，这种解决方法只能提供有限的性能改善。

当共享数据可以被缓存时，实现高速缓存的一致性可采用两种方法。第一种方法是每当一个数据项被修改后，服务器都向所有高速缓存发送无效化消息。第二种方法是仅传播更新。大多数高速缓存系统使用其中一种方法来实现高速缓存的一致性。有时，一些客户服务器数据库动态地选择发送无效化消息或发送更新。

最后，我们还需要考虑进程修改被缓存的数据时可能发生的情况。使用只读高速缓存时，更新操作只能由服务器执行，然后服务器根据某个分发协议来确保更新被传播到各个高速缓存。很多系统使用基于拉式的方法。此时，如果客户检测到它的高速缓存已经过时，它就向服务器请求更新。

另一种方法是允许客户直接修改被缓存的数据，并将更新转发给服务器。**直写式高速缓存**(write-through cache)采用这种方法，分布式文件系统常使用这种直写式高速缓存。实际上，直写式高速缓存类似于基于主备份的本地写协议，客户的高速缓存成为一个临时的主备份。为了保证顺序一致性，客户必须被授予独占写的特权，否则可能发生写写操作冲突。

直写式高速缓存潜在地提供较其他方法更好的性能，这是因为所有的操作都可以在本地执行。如果我们通过允许在通知服务器更新之前执行多个写操作来推迟传播更新，那么性能可以得到进一步的改善。这种方法导致了**回写式高速缓存**(write-back cache)的出现，这种回写式高速缓存也主要应用于分布式文件系统。

7.5.5 实现以客户为中心的一致性

作为一致性协议的最后一个主题，让我们来看看如何实现以客户为中心的一致性。如

果忽略性能问题，实现以客户为中心的一致性相对来说还是简单的。下面我们先描述一下这种实现，然后介绍一个更为实际的实现。

7.5.5.1　简单实现

在以客户为中心的一致性的一种简化实现中，每个写操作 W 都被分配一个全局唯一的标识符。服务器把该标识符赋给提交写操作的客户。在持续一致性中，我们称该服务器为 W 的源。然后，对于每个客户，我们跟踪两个写操作集。客户的读操作集是由客户所执行的读操作相关的写操作组成的。同样，写操作集由客户执行的写操作的标识符组成。

单调读一致性的实现如下。当一个客户在一台服务器上执行一个读操作时，该服务器获得客户的读操作集，检查所有已标识的写操作是否已经在本地执行(这个集的大小可能会引起性能问题，下面将讨论这一问题的解决方案)。如果没有，那么它就联系其他服务器以确保执行读操作之前将本地副本更新。另一种方法是将读操作转发到已经执行了这些写操作的服务器那里执行。读操作执行后，在所选择的服务器上执行的写操作以及与读操作相关的写操作会被加入客户的读操作集。

注意，这里应该可以确定在读操作集中标识的写操作所发生的确切位置。例如，写操作标识符可能包括启动该操作的服务器的标识符。例如，要求那台服务器将写操作记入日志以便在其他服务器上重新执行这个写操作。另外，写操作应该以启动它们的顺序执行。让客户生成一个全局唯一的序列号，并将这个序列号包含于写操作标识符之中，就可以实现写操作的排序。如果每个数据项只能被它的所有者修改，那么后者就可以提供这个序列号。

单调写一致性的实现方式与单调读一致性的实现方式类似。每当一个客户在服务器上启动一个新的写操作，客户的写操作集都会转交到该服务器(同样，由于性能需求，这个写操作集的不能过大。下面将讨论另一种可选择的解决方案)。然后，该服务器确保先执行被标识的写操作，并按正确的顺序执行这些写操作。执行新操作之后，该操作的写标识符将被加入写操作集。注意，使用客户的写操作集更新当前服务器可能会造成客户的响应时间的大幅增加。

同样，写后读一致性要求执行读操作的服务器已经看到客户的写操作集的所有写操作。执行读操作之前，可以简单地从其他服务器获取这些操作，尽管这可能导致较差的响应时间。或者，客户端软件可以查找已经执行了已标识写操作的服务器，写操作在客户的写操作集中标识。

最后，读后写一致性可以通过这一方法实现，即先使用在客户的读操作集中标识了的写操作更新所选择服务器，然后将写操作的标识符以及读操作集中的标识符(这些操作现在已经成为与刚刚执行的写操作相关的操作)加入到写操作集中。

7.5.5.2　提高效率

很容易看出，与每个客户关联的读操作集和写操作集可能变得非常大。为了维持这些集合的可控制性，客户的读操作和写操作根据会话分成组。通常，**一个会话**(session)与一个应用程序关联：应用程序启动时，打开会话，应用程序结束时，关闭会话。但是，会话也可能与暂时结束的应用程序关联，例如用于电子邮件程序的用户代理就是这样的。每当客户结束会话，其关联的集合将被完全清空。当然，如果一个客户打开一个从不关闭的会话时，

与该客户所关联的读操作集和写操作集仍可能变得非常大。

简单实现的主要问题在于如何表示读操作集和写操作集。每个集合由一些写操作的标识符组成。每当客户向服务器转发一个读请求或写请求时，标识符的集合也将被转交给这台服务器，以查看该服务器是否已经执行了所有与该请求相关的写操作。

按照如下方式使用向量时间戳可以更高效地表示这一信息。首先，每当服务器接受一个新的写操作 W 时，它首先按上面所述方法为该操作分配一个全局唯一的标识符和一个时间戳 ts(W)。随后在该服务器上执行的写操作将被分配一个更高值的时间戳。每台服务器 S_i 都维护一个向量时间戳 WVC_i，其中 $WVC_i[j]$ 的值等于 S_i 已接受并执行的、由服务器 S_j 启动的最后一个写操作的时间戳。

每当客户发出在一个特定的服务器上执行读或写操作 O 的请求时，该服务器都返回该操作当前的时间戳和该操作 O 的结果。然后，用向量时间戳表示读操作集和写操作集。对于任何会话 A，我们构造一个向量时间戳 SVC_A，其中 $SVC_A[i]$ 被设置为集合 A 中所有由服务器 S_i 启动的操作的时间戳的最大值。

$$SVC_A[i]=\max\{ts(W)\mid W\in A \ \& \ origin(W)=S_i\}$$

换句话说，会话的时间戳总是表示应用程序所看到的最后一个写操作，其中该应用程序作为会话的一部分正在运行中。使用来自同一个服务器的单个时间戳来表示所有的写操作，就可以进行简化。

另一个示例是，假设作为会话 A 的一部分的客户登录到服务器 S_i。该客户把 SVC_A 传递给 S_i。假设 $SVC_A[j]>WVC_i[j]$。这意味着 S_i 还没有看到来自 S_j 的写操作。根据一致性的要求，服务器 S_i 在回复客户之前必须获取这些写操作。一旦执行了这些操作，服务器 S_i 将返回其当前的时间戳 WVC_i。此时，SVC_A 调整为：

$$SVC_A[j](\max\{SVC_A[j],\ WVC_i[j]\}$$

同样，我们可以看到，在分布式系统中，向量时间戳提供了一种表示历史记录的优雅而紧凑的方法。

本章小结

复制数据主要有两个原因：提高分布式系统的可靠性或提高性能。复制导致了一致性问题：每当一个副本被更新，该副本就变得与其他副本不同了。为了保持各副本一致，我们需要以一种让人注意不到的暂时的不一致方式传播更新。但是，这样做可能会严重地降低性能，在大型分布式系统中尤其如此。

这一问题的唯一解决方法是放松一致性的要求。有多种不同的一致性模型。对持续一致性来说，其目标是设置副本之间的数值偏差、新旧偏差以及操作顺序偏差的界限。

数值偏差指的是副本之间的数值差。这种类型的偏差与应用程序高度相关，例如，可以用在股票记录的副本中。新旧偏差指的是认为某个副本仍为一致性的时间段，尽管在一段时间之前已经发生了更新。新旧偏差经常用于 Web 缓冲。最后，顺序偏差指的是服务器未发送出的暂时写操作的最大数，此时该服务器无需与其他副本服务器同步。

一致性的操作顺序是很多一致性模型的基础。有多种一致性模型，但只有一部分在应用程序开发人员中流行。顺序一致性主要提供的是编程人员在并行程序设计时所期望的语义：每个写操作都是以相同的顺序执行。另一种较少使用的是因果一致性，它反映的是相

互依赖的操作按照其他的依赖关系执行。

弱一致性模型考虑的是读操作和写操作的序列。特别是，它们假定每个序列根据同步变量(例如锁)伴随的操作被适当地“分类”。尽管这要求程序开发人员显式地说明，但是弱一致性模型一般比纯顺序一致性模型更容易以高效的方式实现。

与这些以数据为中心的模型相对，移动用户的分布式数据库领域的研究人员已定义了一系列以客户为中心的一致性模型。这种模型不考虑数据可能被多个用户共享，而是关注单个客户应提供的一致性。其基本设想是，客户在一段时间内要连接不同的副本，而这些不同副本的差别应该是透明的。以客户为中心的一致性模型主要是确保每当客户连接一个新副本时，该新副本使用该客户先前操作的数据更新，而这些数据有可能存储在其他副本处。

要传播更新，可使用不同的技术。不同的技术根据实际传播什么信息、传播到什么地方以及谁启动传播加以区分。我们可以决定传播通知、操作或状态。同样，不是每个副本都需要立即被更新。什么时间更新哪个副本取决于分布式协议。最后，传播更新技术还可以选择更新是被推入到其他副本，还是一个副本从其他副本处拉入更新。

一致性协议描述的是一致性模型的特定实现。按照顺序一致性及其各种变型，一致性协议可分为两种：基于主备份的协议和复制的写协议。在基于主备份的协议中，所有更新操作都被转发到一个主副本，然后该主副本确保更新被正确地排序和转发。在复制的写协议中，更新同时被转发到多个副本。在这种情况下，正确地对操作进行排序变得更加困难。

习　题

1. 对共享的 Java 对象的访问可以通过将其方法声明为同步的而被串行化。当这种对象被复制时，这种方法足以保证访问的串行化吗？
2. 请用自己的语言解释实际考虑弱一致性模型的主要原因。
3. 请解释 DNS 是如何进行复制的，以及它实际运行很好的原因。
4. 在讨论一致性模型时，经常提及软件和数据存储间的合约。为什么需要这一合约？
5. 给定图 7.2 的副本，应如何最终化一致性单元中的数值，使得 A 和 B 可以看到相同的结果。
6. 在图 7.7 中，000000 是顺序一致性存储器的合法输出吗？请解释您的答案。
7. 通常认为弱一致性模型给程序开发人员增加了额外的负担。这一命题在哪种程度上确实是正确的？
8. 使用序列器并为维持主动复制的一致性而进行的全序的广播是否违背系统设计中的端到端理论？
9. 您会选择哪种类型的一致性来实现电子股票市场？请解释您的答案。
10. 如果要将一个移动用户的个人邮箱作为广域范围的分布式数据库的一部分来实现，哪种类型的以客户为中心的一致性最为合适？
11. 请描述一个用于显示刚被更新的 Web 页面的写读一致性的简单实现。
12. 为简单起见，我们假设在 Bayou 中不存在写写冲突。当然，这是不切实际的假设。请解释冲突是如何发生的？
13. 使用租用时，客户和服务器的时钟分别紧密同步是否是必要的？
14. 我们已介绍过，使用 Lamport 逻辑时钟的全序多播不能扩展。请解释原因。

15. 请证明，在持续一致性情况下，
16. 在介绍持续一致性时，我们假设每个写操作只增加数据项 x 的值。请描述一种也可以减少 x 的值的解决方法。
17. 考虑一个用于保证分布式数据存储上顺序一致性的非阻塞的主备份协议。这样的数据存储总能提供写读一致性吗？
18. 为了使主动复制正常工作，所有操作必须以相同的顺序在每个副本处被执行。这个顺序总是必需的吗？
19. 为了使用序列器实现全序的广播，一种方法是先将操作转发给序列器，然后该序列器为这个操作分配一个唯一的号码，随后广播这个操作。请提出另外两种可选择的方法，并比较这三种方法。
20. 一个文件被复制在 10 个服务器上。请列出表决算法允许的所有读团体和写团体。
21. 通过让服务器与尽量少的客户保持联系，就可以用基于状态的租用来减少该服务器的负载。该方法能得到更好的性能吗？
22. (**实验**)作为一个练习，请你实现一个支持多播 RPC 的简单系统。我们假设系统有多个复制的服务器，每个客户可以通过 RPC 与一个服务器通信。但是，处理复制时，客户需要向每个副本发送一个 RPC 请求。设计客户程序，使得客户好像只往应用程序发送单个的 RPC。假设复制的目的是为了提高性能，而那些服务器可能是易于出故障的。

第8章

容错性

分布式系统区别于单机系统的一个特性是它可以容许部分失效。当分布式系统中的一个组件发生故障时就可能产生部分失效。这个故障也许会影响到其他组件的正确操作，但同时也有可能完全不影响其他组件。而非分布式系统中的故障通常会影响到所有的组件，可能很容易就使整个应用程序崩溃。

分布式系统设计中的一个重要目标是以这样的方式来构造系统：它可以从部分失效中自动恢复，而且不会严重地影响整体性能。特别是，当故障发生时，分布式系统应该在进行恢复的同时继续以可接受的方式进行操作，也就是说，它应该能容许错误，在发生错误时某种程度上仍可以继续操作。

在本章中，我们将更详细地学习使分布式系统具有容错性的技术。在提供了一些有关容错的背景知识后，我们将介绍进程恢复与可靠多播。进程恢复涉及到的技术可以使一个或多个进程发生故障而不会严重影响到系统的其余部分。可靠多播与这个问题相关，它确保可以成功地把信息传输到一个进程集合。要保持进程同步，可靠多播通常是必需的。

原子性(atomicity)在很多应用程序中都是一个重要的特性。例如，在分布式事务中，就必须要保证一个事务中的所有操作全部发生或一个都不发生。在分布式系统中原子性的基础是分布式提交协议，我们将在本章中用单独一节来讨论它。

最后，我们将介绍如何从故障中恢复。特别是，我们将考虑何时以及如何对一个分布式系统的状态进行保存以确保在以后恢复到该状态。

8.1 容错性概述

容错涉及到计算机科学中的很多研究。在本节中，我们将从介绍与容错相关的基本概念开始，然后讨论故障的模式。我们也要讨论处理故障的关键技术——**冗余**(redunsancy)。要获得关于分布式系统中有关容错的更多信息，请参阅文献(Jalote 1994)或(Shooman 2002)。

8.1.1 基本概念

要理解容错在分布式系统中的作用，我们首先需要深入地了解分布式系统中的容错到

底意味着什么。容错与被称为**可靠的系统**(dependable system)紧密相关。可靠性是一个术语,它包含了分布式系统中很多有用的需求,列举以下(Kopetz and Verissimo 1993):

(1) 可用性;

(2) 可靠性;

(3) 安全性;

(4) 可维护性。

可用性(availability)被定义为系统的一个属性,它说明系统已准备好,马上就可以使用。通常,它指在任何给定的时刻,系统都可以正确地操作,可根据用户的行为来执行它的功能。换句话说,高度可用的系统在任何给定的时刻都能及时地工作。

可靠性(reliability)是指系统可以无故障地持续运行。与可用性相反,可靠性是根据时间间隔而不是任何时刻来进行定义的。高度可靠的系统可以在一个相对较长的时间内持续工作而不被中断。这很微妙,但是与可用性相比,这是一个重要的不同。如果系统在每小时中崩溃 1ms,那么它的可用性就超过 99.9999%,但是它还是高度不可靠的。与之类似,如果一个系统从来不崩溃,但是要在每年 8 月中停机两个星期,那么它就是高度可靠的,但是它的可用性只有 96%。这两种属性并不相同。

安全性(safety)是指在系统偶然出故障的情况下能正确操作而不会造成任何灾难。例如,很多进程控制系统,比如那些用来控制核电站或把人送入太空的控制系统,就必须提供高度的安全性。这样的控制系统即使只是非常短时间瞬时故障,结果也将是灾难性的。过去的很多例子(以后还会有更多的例子)都说明了要建立安全的系统是多么困难。

最后,**可维护性**(maintainability)是指发生故障的系统被恢复的难易程度。高度可维护的系统可能具有高度的可用性,特别是在可以探测到故障并自动恢复时。但是,像我们在本章后面将看到的那样,从故障中自动恢复说起来简单,做起来就难了。

通常也要求可靠的系统提供高度的安全性,特别是在要处理诸如完整性这样的问题时。我们将在下一章中讨论安全问题。

当一个系统不能兑现它的承诺时就被认为是**失败**(fail)了。尤其是,如果一个分布式系统被设计为为它的用户提供大量的服务,当这些服务中的一个或多个不能被(完整地)提供时,系统就失败了。而**错误**(error)是系统状态的一部分,它可能会导致失败。例如,当从网络上传输数据包时,可能有一些数据包在到达接收方时已经被破坏了。在这种环境下,被破坏了意味着接收方可能错误地得到某位的值(例如把 0 读成 1),甚至无法探测到有什么东西到达了。

造成错误的原因被称为**故障**(fault)。无疑,找到是什么引起了错误是很重要的。例如,不好的传输介质可能很容易使得数据包被破坏。在这种情况下要解决故障是相对容易的。但是无线网络中的传输错误可能是由恶劣的天气条件引起的。要改变天气来减少或防止错误是不可能的。

建立一个可靠的系统与控制故障紧密相关。防止、解决和预报故障三者之间是有差别的(Avizienis 等 2004)。对我们来说最重要的问题是**容错**(fault tolerance),它意味着系统即使在发生故障时还能提供服务。

故障通常被分为暂时的、间歇的和持久的。**暂时故障**(transient fault)只发生一次,然后就消失了,即使重复操作也不会发生。一只鸟从微波传输的电波中飞过可能会使一些网

络上的数据丢失。如果传输超时重发,第二次就会正常工作。

间歇故障(intermittent fault)是发生,消失不见,然后再次发生,如此反复进行。连接器接触不良通常会造成间歇故障。间歇故障会造成情况的恶化,因为它们很难诊断,通常,当解决故障的人到来时系统工作良好。

持久故障(permanent fault)是那些直到故障组件被修复之前持续存在的故障。芯片燃烧、软件错误和磁盘头损坏都是持久故障的例子。

8.1.2 故障模式

发生故障的系统不能充分地提供所设计的服务。如果我们把分布式系统视为一个彼此之间且与它们的客户进行通信的服务集,那么不能充分提供服务就意味着服务器、通信通道或两者都不能正常地进行工作。但是,出现故障的服务器本身并不总是我们应该查找错误的地方。如果这样的服务器要依赖其他服务器才能充分提供它的服务,那么错误的原因就可能需要到别的地方去寻找。

这样的依赖关系大量出现在分布式系统中。如果一个文件服务器的设计目的是提供高度可用的文件系统,那么失效的磁盘会使该服务器运行困难。如果这样的文件服务器是分布式数据库的一部分,那么整个数据库的正常工作都将被打乱,因为只有一部分数据真正可以访问。

为了更好地理解故障到底有多严重,人们开发了一些分类方法。其中一种方法如图 8.1 所示,它基于(Cristian 1991)和(Hadzilacos 和 Toueg 1993)中描述的方法。

故障类型	说明
崩溃性故障	服务器停机,但是在停机之前工作正常
遗漏性故障	服务器不能响应到来的请求
接收故障	服务器不能接收到来的消息
发送故障	服务器不能发送消息
定时故障	服务器的响应在指定的时间间隔之外
响应故障	服务器的响应不正确
值故障	响应的值错误
状态转换故障	服务器偏离了正确的控制流
随意性故障	服务器可能在随意的时间产生随意的响应

图 8.1 不同类型的故障

如果服务器过早停机但在停止之前工作正常,就发生了**崩溃性故障**(crash failure)。崩溃性故障的一个重要方面是,一旦服务器停机,就不再提供任何服务。崩溃性故障的一个典型例子是操作系统崩溃,这时只有一种解决方法:重新启动。虽然人们期望个人计算机系统能正常运行,但它经常遭遇崩溃性故障。在这个意义上说,把复位按钮从机箱背后移到前面是很有道理的。可能有一天还会把它移到背后,甚至完全去掉。

当服务器不能对请求进行响应时就发生**遗漏性故障**(omission failure)。导致这种问题可能有多种原因。在发生接收遗漏性故障的情况下,首先服务器可能永远不会接收到请求。注意,这可能是由于尽管在客户和服务器之间正确地建立起连接,但是没有线程监听到来的

请求。接收遗漏性故障通常不会影响到服务器的当前状态，因为服务器不知道有信息发送给了它。

与之类似，如果服务器能正常工作，但是在发送响应时失败，则产生发送遗漏性故障。例如，当发送缓冲区溢出而服务器又没有为这样的情况做好准备时就发生此类故障。注意，与接收遗漏性故障，服务器现在的状态可能说明它已经完成了对客户的服务。因此，如果响应发送失败，那么服务器可能需要为客户重新发送先前的请求而做好准备。

另外一种遗漏性故障与通信无关，它可能由诸如无限循环或不正确的内存管理的软件错误引起，这时服务器的状态被称为“挂起”。

另外一种故障类型与时间有关。如果响应是在指定的实时间隔之外，就发生了**定时故障**(timing failure)。像我们在第2章中介绍的同步数据流那样，如果提供数据的速度过快，而在接收方又没有足够的缓冲空间来保存所有到来的数据，那么就很容易在接收方引起问题。但是，更通常的情况是服务器的响应太慢，这种情况被称为发生了“性能故障”。

一种严重的故障类型是**响应故障**(response failure)，就是说服务器的响应不正确。可能发生两种响应故障。在发生数值故障的情况下，服务器为请求提供错误的响应，例如搜索引擎系统返回了与使用的搜索项无关的Web页面，就是这种情况。

另外一种响应故障被称为**状态转换故障**(state transition failure)。当服务器对到来的请求做出意想不到的响应时就发生这种故障。例如，如果服务器接收到一个它不能识别的信息，也没有采取措施来处理这样的信息，那么就会发生状态转换故障。特别是，发生故障的服务器可能会错误地采取一种从来没有初始化的默认行为来进行处理，此时就发生这种故障。

最严重的故障是**随意性故障**(arbitrary failure)，也被称为**拜占庭故障**(Byzantine failure)。实际上，当发生随意性故障时，客户应该做好最坏的准备。特别是，服务器可能产生它从来没有产生过的输出，但是又不能检测出错误。更坏的情况是发生故障的服务器恶意地与其他服务器共同工作来产生恶意的错误结果。这种情况说明了为什么谈到可靠系统时安全被认为是一个重要的需求。术语“拜占庭”是指拜占庭帝国，它存在的时间是330到1453年，地点在巴尔干半岛和现在的土耳其，当时在统治阶级中充斥着无休止的阴谋诡计和谎言。拜占庭故障的问题由Pease等(1980)和Lamport等(1982)首先进行了分析。我们将在下面对此类故障进行分析。

随意性故障与崩溃性故障紧密相关。崩溃性故障的定义在上面已经给出，它是服务器停机的最“好”方式。它也被称**为失败停机故障**(fail-stop failure)。实际上，因故障停机的服务器只是简单地停止产生输出，而且它的停机能够被其他的进程探测到。例如，服务器可以“友好地”宣布它将要崩溃。

当然，在现实中，服务器通过呈现遗漏性故障或崩溃性故障而停机，它们并不“友好地”宣布将要停机，而要等其他的进程来判定服务器是否已经过早停机。但是，在这样的**失败沉默系统**(fail-silent failure)中，其他的进程可能错误地认为服务器已经停机，而服务器只不过是意外地慢，也就是说发生了性能故障。

最后，服务器也有可能产生随机的输出，但是这个输出能简单地被其他进程识别为垃圾。这时服务器以一种“仁慈”的方式来出现随意性故障，这些故障也被称为**安全失败**(fail-safe)。

8.1.3　使用冗余掩盖故障

如果系统是容错的，那么它能做的最好的事情就是对其他进程隐藏故障的发生。关键技术是使用冗余来掩盖故障。有三种可能：信息冗余、时间冗余和物理冗余，请参阅文献(Johnson 1995)。在信息冗余中，添加额外的位可以使错乱的位恢复正常。例如可以在传输的数据中添加一段 Hamming 码来从传输线路上的噪声中恢复数据。

在时间冗余中，执行一个动作，如果需要就再次执行。使用事务处理(在第 1 章中已经介绍)是这种方法的一个例子。如果一个事务处理中止，那么它就可以无害地重新执行。当错误是临时性或间歇性时，时间冗余特别有用。

在物理冗余中，通过添加额外的装备或进程使系统作为一个整体来容忍部分组件的失效或故障成为可能。物理冗余可以在硬件上也可以在软件上进行。例如，可以在系统中添加额外的进程，这样如果少数进程崩溃，系统还是可以正常工作。换句话说，通过冗余的进程可以获得高度容错性。

物理冗余是提供容错性的著名技术。它用在生物学(哺乳动物具有两只眼睛、两个耳朵、两个肺等)、飞行器(747 具有四个引擎，而只使用三个)和体育(有多个裁判)中。物理冗余用在电子电路的容错中已经有多年了，了解如何在电子电路中使用它是有意义的。例如，考虑图 8.2(a)中的电路。这里的信号依次通过设备 A、B 和 C。如果它们中的一个发生故障，最后结果就可能是错误的。

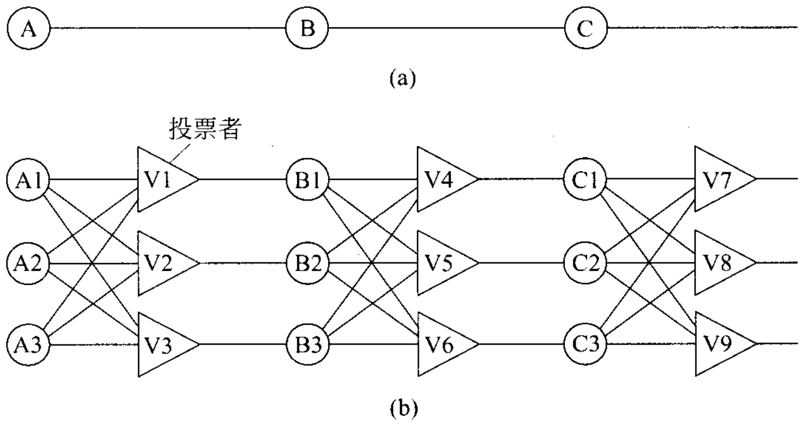


图 8.2　三倍模块冗余

在图 8.2(b)中，每个设备都被复制了三个。这样电路中的每一级都有三个表决电路，每个表决电路都是具有三个输入和一个输出的电路。如果输入中的两个或三个是相同的，那么输出就等于输入，如果所有的输入都是不同的，那么输出就是未定义的。这种设计被称为**三倍模块冗余**(Triple Modular Redundancy，TMR)。

假定元件 A2 发生故障，则每个表决电路 V1、V2 和 V3 获得两个好的(一致的)输入和一个无用的输入，它们中的每个都将正确值输出到第二级。这样 A2 故障的影响就完全被掩盖了，到 B1、B2 和 B3 的输入就与没有发生故障时完全相同。

现在考虑一下如果除了 A2 之外 B3 和 C1 也都发生了故障时的情形。即使发生这种情况，其影响也会被掩盖，三个最后的输出还是正确的。

为什么在每一级都需要三个表决电路呢？毕竟一个表决电路就可以发现并传递主要的观点。但是，一个表决电路只是一个组件，它可能发生故障。例如，假定V1发生故障，到B2的输入就是错误的，但是只要别的部分正常工作，B2和B3就会产生相同的输出，V4，V5和V6都会产生到第三级的正确结果。V1发生故障的影响与B1故障没什么区别，在两种情况下B1都产生错误的输出，但是两种情况下都在后面才进行表决。

尽管不是所有的容错分布式系统都使用TMR，但是这种技术还是很普遍的，它对容错系统是什么给出了一个清楚的解释。它与使用高度可靠的单个组件的系统不同，后者的组织不是容错的。当然，TMR也可以递归使用，例如在芯片内部使用TMR来使它高度可靠，而对于使用芯片的电路设计者来说，他们并不知道芯片的冗余性。

8.2 进程恢复

已经讨论了容错的基本问题，现在我们集中讨论在分布式系统中如何获得容错性的问题。我们首先要讨论的主题是防止进程失败，这是通过把进程复制到组中来获得的。在下面的内容中，我们考虑进程组中的一般设计问题，并讨论容错组到底是什么。我们也将讨论当一个或多个程序不能给出正确的回答时，如何在进程组中达到一致。

8.2.1 设计问题

容忍失败进程的关键方法是把多个同样的进程组织到一个组中。所有组都具有的关键特性是当信息发送到组本身时，组中的所有成员都接收它。通过这种方式，如果组中的一个进程失败，其他的一些进程可以接管它(Guerraoui 和 Schiper 1997)。

进程组可以是动态的。可以创建新的组也可以删除旧的组。在系统操作过程中，一个进程可以加入一个组也可以离开一个组。一个进程可以同时是多个组的成员。因此需要一些机制来管理组和组的成员。

组类似于社会组织。Alice可能是一个读书俱乐部、一个网球俱乐部和一个环境组织的成员。在一个特别的日子里，她可能从读书俱乐部收到一本生日蛋糕食谱的邮件(消息)，从网球俱乐部收到每年一度的母亲节网球锦标赛的消息，还从环境组织收到保护南方土拨鼠运动开始的消息。在任何时刻，她都可以自由地离开任何一个或所有这些组，也可能加入其他的组。

引入组的目的在于允许把进程的集合作为单一的抽象概念来处理。这样，一个进程就可以把消息发送给一个服务器组而不用知道有多少个进程以及它们在哪里，而这些可以在两次调用之间进行改变。

8.2.1.1 平等组与等级组

不同的组之间一个重要的区别在于它们的内部结构。在一些组中，所有的进程都是平等的。没有指挥，所有的决定都是共同做出的。在其他的组中存在一些等级关系。例如，一个进程是协作者而其他进程都是工作者。在这种模式中，当外部客户或一个工作者产生一

个工作请求时，请求被发送给协作者。协作者决定哪个工作者最适合，然后把请求转发给它。当然也可能有更复杂的等级关系。在图 8.3 中说明了这些通信模式。

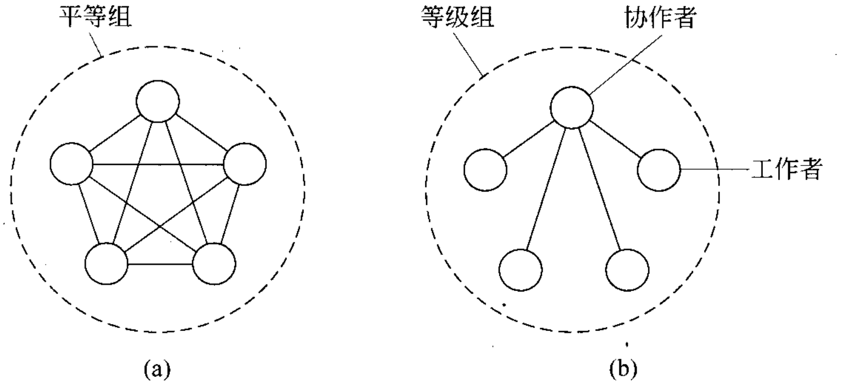


图 8.3 平等组中和简单等级组中的通信

(a) 平等组中的通信；(b) 简单等级组中的通信

每种组织结构都具有它的优点与缺点。平等组是对称的，没有单独的失败点。如果一个进程崩溃，组只是简单地变得更小，但是还可以继续。它的缺点在于做出决定比较复杂。例如，要决定某事常常需要进行表决，这导致了一些延迟和开销。

等级组则相反。协作者的故障会使整个组崩溃，但是只要它保持运行，就可以独自做出决定，不需要其他进程参加。

8.2.1.2 组成员

当组通信发生时，需要一些方法来创建和删除组，以及允许进程加入和离开组。一种可能的方法是使用**组服务器**(group server)，所有这些请求都发送给它。这个组服务器保持着所有组及其成员的完整的数据库。这种方法直接、有效、容易实现。不幸的是它具有所有集中式技术的缺点：单一的失败点。如果组服务器崩溃，组管理就不再存在。可能大多数或所有组都不得不从头开始重建，所有正在进行的工作都可能终止。

与之相反的方法是以分布式的方法来管理组成员。例如，如果可靠的多播可用，一个外部进程就可以发送消息给所有的组成员表示它希望加入该组。

要离开一个组，理想的情况是，成员只需要给所有成员发送一个再见消息。在容错环境中，故障停机的语义一般并不确切。问题在于当一个进程崩溃时不能像一个进程自动离开那样进行通知。其他成员不得不通过注意到崩溃的成员不再进行响应来发现这一点。一旦确定崩溃的成员是真正的崩溃(而不是速度慢)，就从组中删除它。

另外一个棘手的问题是进程的离开与加入必须跟数据消息的发送同步。换句话说，从进程加入一个组开始，它就必须接收发送给该组的所有消息。与之类似，一旦进程离开一个组，它就不能接收来自该组的任何消息，其他的成员也不能接收来自它的任何消息。一种确保把进程的加入或离开集成到消息流中的正确位置的方法，是把这个操作转换为一个发送到整个组的消息序列。

与组成员相关的最后一个问题是，如果很多机器停机使得组完全不能工作时该怎么办。这时需要通过一些协议来重建组。一些进程不得不进行初始化来重新启动，但是如果有两个或三个进程同时进行尝试会发生什么？协议必须能够处理这些问题。

8.2.2 故障掩盖和复制

进程组是建立容错系统方法的一部分，特别是在有一组相同的进程允许我们掩盖组中的一个或多个发生故障的进程的情况下。换句话说，我们可以复制进程并把它们组织在一个组中来用一个容错的组取代一个脆弱的进程。像前面讨论过的那样，有两种方法来进行复制：通过基于主进程的协议或通过复制写协议。

在容错的情况下，基于主进程的复制通常以主进程后备协议的形式出现。在这种情况下，以等级方式来组织一个进程组，其中一个主进程协调所有的写操作。在实践中，这个主进程是固定的，尽管如果需要可以用一个后备进程来接管它。实际上，如果主进程崩溃，后备进程执行一些选举算法来选择一个新的主进程。

像我们在上一章中解释的那样，与基于团体的协议相同，复制写协议以主动复制的形式使用。这些解决方法把相同进程的集合组织到一个平等组中。这样做的主要优点在于这样的组没有单一的失败点，代价是分布式的协调。

使用进程组进行容错中一个重要的问题是需要多少复制。为了简化我们的讨论，我们只考虑复制—写系统的情况。如果系统能够经受 k 个组件的故障并且还能满足规范的要求，那么就被称为 **k 容错**(k fault tolerant)。如果这些组件(进程)是失败沉默的，那么具有 k+1 个组件就足够提供 k 容错。如果 k 个组件停止工作，还可以用剩下的一个组件来得到响应。

另一方面，如果进程发生拜占庭失败，继续错误运行并发送出错误或随机的应答，那么最少需要 2k+1 个进程才能获得 k 容错。在最坏的情况下，k 个失败的进程可以意外地(甚至是故意地)产生同样的应答，但是剩余的 k+1 个进程也会产生同样的回答，这样，客户或表决电路还可以相信多数进程的回答。

当然，说一个系统是 k 容错的，让 k+1 个相同的进程以票数胜过 k 个相同的应答在理论上是好的，但是在实践中很难想象这样的情况，即 k 个进程失败了，但是 k+1 个进程没有失败。这样，即使在容错系统中也需要某种统计分析。

与这种模式有关的一个不明显的前提是所有请求按相同的顺序到达所有的服务器，这个问题被称为**原子多播问题**(atomic multicast problem)。实际上，这个条件可以稍微放松，因为读操作不会有问题而一些写操作也可以替换，但是还留有一般的问题。原子多播将在后面详细讨论。

8.2.3 故障系统的协定

把复制的进程组织在一个组中有助于提高容错性。正如我们提到的那样，如果客户可以根据它的表决机制做出决定，我们甚至可以容忍 2k+1 个进程中的 k 个进程得出错误的结果。但是，我们做出的假定是这些进程没有组合成团队来共同产生错误的结果。

通常，如果我们要求进程组达成一致，事情会变得更复杂。在很多情况下需要达到一致，例如，选择一个协作者，决定是否提交一个事务，在工作者之间划分任务以及同步。当通信和进程都很完美时，达成这样的一致通常是简单而又直接的，但是如果不是这样，就会出

现问题。

分布式协议算法的一般目标是使所有的非故障进程就一些问题达成一致，而且在有限的步骤内就达成一致。这个问题有点复杂了，因为底层系统的不同假设需要不同的解决方案(假设解决方案存在)Turek 和 Shasha(1992)区分了以下一些情况：

(1) 同步与非同步系统。在某个加锁步骤模式下，有且只有一个进程运行，该系统就称为同步的(synchronous)。正式地说，这表示，有一常量 $c \geqslant 1$，如果有某个处理器执行了 $c+1$ 步，其他进程最少执行 1 步。不是同步的系统就称为是**异步的**(asynchronous)。

(2) 通信延时受限与否。当且仅当每个消息使用全局和预定的最大时间传送时，延时是受限的。

(3) 消息传送是否已排序。

(4) 消息传输是通过单播或多播来完成的。

事实证明，只有在如图 8.4 所示的情况下才可能达成协定。在其他所有情况下，都不存在解决办法。注意，大多数实际的分布式系统都是假定进程是异步的，消息传输是单播的，通信延时是不受限的。因此，我们需要使用已排序(可靠)的消息传送，如 TCP/IP。图 8.4 显示了当进程发生故障时的分布式协定非凡本性。

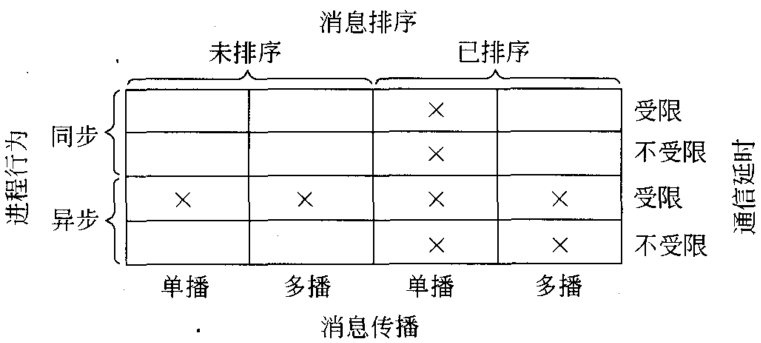
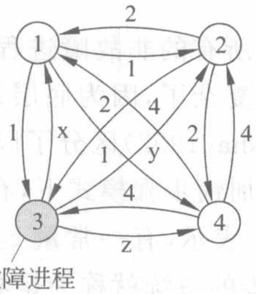


图 8.4 分布式协定下能够达到的环境

这个问题最初是由 Lamport 等(1982)开始研究的，又称为**拜占庭协定问题**(Byzantine agreement problem)，意思是在众多战争中，多个军队需要达成的协定，例如，面临反叛将军、谋反中尉等情况下的军队战斗力。请看下面由 Lamport 等(1982)描述的解决办法。在这种情况下，我们假定进程是同步的，消息是单播的，且是按顺序进行的，通信延时是有限制的。我们假定有 N 个进程，每个进程为其他进程提供一个值 v_i。目标是让每个进程构建一个长度为 N 的向量，这样，如果进程 i 是无故障的，那么 $V[i]=v_i$。否则，V[i]就是未定义的。

在图 8.5 中显示了 $N=4$ 和 $k=1$ 情况下的算法。对这些参数，该算法分四步操作。第一步，每个无故障进程 i 使用可靠单播给其他每个进程发送 v_i。故障进程则可能发送些其他内容。而且，由于我们使得的是多播，它们可能会给不同的进程发送不同的值。假设 $v_i=i$。在图 8.5(a)中，我们可以看到，进程 1 给其他进程发送的是 1，进程 2 发送的是 2，进程 3 发送的分别是 x、y 和在，进程 4 发送的是 4。在第二步中，把第一步中的结果集合成图 8.5(b)所示的向量形式。

第三步，每个进程把图 8.5(b)的向量发送给其他进程。此时，每个进程得到 3 个向量，



(a)

1 Got(1, 2, x, 4)		
2 Got(1, 2, y, 4)		
3 Got(1, 2, 3, 4)		
4 Got(1, 2, z, 4)		

(b)

1 Got	2 Got	4 Got
(1, 2, y, 4)	(1, 2, x, 4)	(1, 2, x, 4)
(a, b, c, d)	(e, f, g, h)	(1, 2, y, 4)
(1, 2, z, 4)	(1, 2, z, 4)	(i, j, k, l)

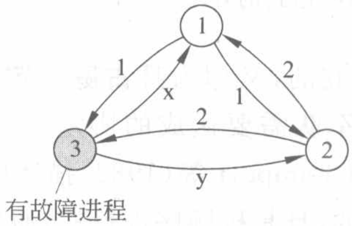
(c)

图 8.5　3个无故障进程和1个有故障进程的拜占庭协定问题

(a) 每个进程给其他进程发送值；(b) 每个进程构建的向量；(c) 每个进程在第三步中接收的向量

每个向量来自其他3个进程。这里，由于进程3是有故障进程，产生了从a到l共12个新值。第三步的结果如图8.5(c)所示。最后，在第四步中，每个进程检查每个新接收向量中的第i个元素。如果值占大多数，那么该值就放置到结果向量中。如果值不占多数，那么结果向量的响应元素就标记为未知(UNKNOWN)。从图8.5(c)可以看出，1、2和4都与v_1、v_2和v_4的值一致，是正确的结果。从这些进程得出的结论无法确定v_3，但也是不相关的。拜占庭协定的目标是一致性意见只与无故障进程的值有关。

下面让我们来看看N=3和k=1的情况，也就是说，只有2个无故障进程和1个有故障进程，如图8.6所示。这里我们可以看到，在图8.6(c)中，无故障进程的向量中无法看到元素1、2和3占大多数的情况，因此都标记为未知。该算法不能产生协定。



(a)

1 Got(1, 2, x)		
2 Got(1, 2, y)		
3 Got(1, 2, 3)		

(b)

1 Got	2 Got
(1, 2, y)	(1, 2, x)
(a, b, c)	(d, e, f)

(c)

图 8.6　与图8.5相同，不过这里只有2个无故障进程和1个有故障进程

在Lamport等(1982)的文章中证明了，在一个含有k个有故障进程的系统中，只有有2k+1个无故障进程时才可以达到协定，即总进程为3k+1个。用一般术语表示就是，只有无故障进程数多于三分之二时才可以达成协定。

解决这个问题的另一种方法如下。基本上，我们需要获得的是无故障进程组的大多数投票，不管它们之中是否有有故障进程。如果有k个有故障进程，我们需要确保它们的投票和其他所有无故障进程的投票，仍然符合无故障进程占大多数。如果有2k+1个无故障进程，只要有多于三分之二的无故障进程，就可以到达协定。换句话说，如果多于三分之二的进程达成同一决定，这个决定就符号无故障进程组的大多数投票。

但是，达到协定可能会很糟糕。Fischer 等(1985)证明，在分布式系统中，如果不能保证消息在一直且有限的时间内传送出去，即使某个进程是有故障的(纵使某个进程是悄然失败的)，也不可能达成协定。这种系统的问题在于，任意缓慢的进程与崩溃的进程是无法区分的(即，你无法区分进程是死亡还是生存的)。有很多其他的理论结果可以知道协定何时是可能的，何时是不可能的。有关这些内容的介绍请参见(Barborak 等 1993)，以及(Turek 和 Shasha 1992)。

应该注意的是，到目前为止所讨论的方法都是假定结点是拜占庭或协作的。当进程是来自不同管理域时，就不能简单地假定后者了。在这种情况下，它们很可能展示的是理性行为，例如，报告超时比运行一个更新操作更合算。如何处理这种情况并不简单。这种解决方法的第一步是以 **BAR 容错**(BAR fault tolerance)的方式进行，其中，BAR 表示的是 Byzantine、Altruism 和 Rationality。有关 BAR 容错的内容请参见(Aiyer 等 2005)。

8.2.4 故障检测

从我们到目前为止的讨论中可以清楚地看出，要很好地屏蔽故障，通常需要检测出故障。在分布式系统中，故障检测是容错的基石。所有这些都归结为，对于一组进程，无故障的成员应该可以知道谁仍是组成员，谁不是。换句话说，当某个成员出现故障时，我们需能够检测出来。

要检测进程故障，只有两种方法。进程主动地往其他进程发送"are you alive?"消息(它们显然期望有回答)，或者被动地等待来自其他进程的消息。只有在能保证进程间有足够的通信时，后一种方法才有意义。实际上，通常使用的是主动 **pinging**。

关于故障检测的理论工作已经开展了很多。所有这些可以归结为，使用超时机制来检测某个进程是否发生故障。在实际的设置中，这种方法有两个主要问题。首先，由于网络的不可靠，如果仅仅因为没有对 ping 消息返回应答，就认为进程发生了故障，有时候可能是不准确的。如果因为一个错误的判断而把一个正常的进程删除了，显然我们就犯错误了。

另一个严重的问题是，通过超时来判断进程故障很粗糙。正如(Birman 2005)所介绍的那样，在构建恰当的故障检测子系统时，都只是考虑对单个消息没有响应，而没有做更多的工作。看看企业部署的分布式系统就知道，上面说法就更明显了。

在设计故障检测子系统时，需要考虑很多问题(Zhuang 等 2005)。例如，每个结点会定期地告知其邻结点，它仍是活动的，但此信息传播中也会发生故障。因此，正如我们前面介绍过的那样，一种办法是让每个结点主动地探测其他结点。

故障检测也可以作为相邻结点间常规信息交换的边界效应来完成。这种方法也被 Obduro(Vogels 2003)所采用：进程定期地告知它们的服务可用性。这些信息通过网络逐级扩散。最后，每个进程都知道其他进程，更为重要的是，这样就有了足够的信息来确定某个进程是否发生了故障。如果某个成员的可用性信息很旧，旧可假定它发生了故障。

另一个重要的问题是，故障检测子系统应该能很好地区分网络故障和结点故障。处理这个问题的一个办法是，不让某个结点去确定它的某个邻结点是否崩溃了，而是当知道某个 ping 消息超时时，结点请求其他邻结点去查看是否可以到达已假定发生故障的结点。当然，正确的信息也可以共享：如果某个结点还是活动的，那么这种信息就可以转发给其他对

该信息感兴趣的结点。

这就给我们带来了另一个关键问题：当检测发现某个成员发生了故障时，如果告知其他无故障的进程？在 FUSE 中提出了一个简单但基本的方法（Dunagan 等 2004）。在 FUSE 中，进程可以加入到某个组中，而这些进程可以是分布在广域网中的。组成员可以创建一个生成树，用于监视成员故障。成员可以往其邻居发送 ping 消息。如果某个邻成员没有响应，该结点立即转换为不再响应其他结点的 ping 消息的状态。通过迭代，单个结点的故障就可以迅速地提升为组故障通知。FUSE 不受链路故障的影响，其简单的原因是，它依赖的是组成员之间的点对点 TCP 连接。

8.3 可靠的客户-服务器通信

在很多情况下，分布式系统中的容错关注的是故障进程。但是我们也需要考虑通信故障。前面讨论过的大多数故障模型都很好地应用到通信通道上。特别是，通信通道也可能出现崩溃性故障、遗漏性故障、定时故障和随意性故障。在实践中建立可靠的通信通道时，焦点应该放在掩盖崩溃性故障与遗漏性故障上。随意性故障可能以重复消息的形式发生，造成它的原因是在计算机网络中可能把消息缓冲一个相对较长的时间，然后在原始发送方发出重新传送命令时把消息重新发到网络中（Tanenbaum 2003）。

8.3.1 点对点通信

在很多分布式系统中都使用像 TCP 这样可靠的传输协议来建立可靠的点到点通信。TCP 可以通过确认和重传来掩盖遗漏性故障。这样的故障对 TCP 客户是完全隐藏的。

但是通常不能掩盖连接的崩溃性故障。崩溃性故障发生在不论什么原因 TCP 连接突然中断时，这时就不能再通过该通道传输更多的消息了。在大多数情况下，可通过抛出一个异常信号来通知客户通道已经崩溃。要掩盖这样的故障，唯一的方法就是让分布式系统尝试自动建立一个新的连接。

8.3.2 失败时的 RPC 语义

现在我们来看一下在使用像远程过程调用（RPC）或远程方法调用（RMI）这样的高级通信工具时的客户-服务器通信。在下面内容中，我们主要讨论 RPC，但是这些讨论也同样适用于与远程对象的通信。

RPC 的目标是通过使远程过程调用与本地过程调用看上去相同来隐藏通信。除了一些例外情况之外，现在我们已经对它相当熟悉了。确实，只要客户和服务器都正常工作，那么 RPC 就可以很好运转。当发生错误时会出现一些问题。然后就不容易掩盖本地调用与远程调用之间的区别了。

为了使我们的讨论结构化，首先区分一下在 RPC 系统中发生的 5 种失败形式，如下所示：

(1) 客户不能定位服务器；

(2) 客户到服务器的请求消息丢失；

(3) 服务器在收到请求之后崩溃；

(4) 从服务器到客户的响应消息丢失；

(5) 客户在发送请求之后崩溃。

每种类型的失败都会引起不同的问题，需要用不同的方法来解决。

8.3.2.1 客户不能定位服务器

首先是客户可能不能定位适当的服务器。例如，服务器可能因发生故障而停止运行。作为一种选择，我们假设客户程序是使用特定的客户存根版本进行编译的，而且在相当长的时间里不使用二进制。同时服务器程序有了改进，安装了一个接口的新版本，产生了新的存根并投入使用。当客户最终运行时，绑定器可能不能使它与服务器相匹配并报告失败。这种机制用来保护客户不会意外地试图与那些与它所要求的参数以及所假定的动作并不相符合的服务器进行通信，关于如何处理这种失败的问题仍然存在。

一种可能的方法是让错误抛出一个**异常**(exception)，在一些语言中(例如 Java)，程序员可以编写只有在发生特定错误(例如被零除)时才调用的过程。在 C 中可以使用信号处理器。换句话说，我们可以定义一种新的信号类型 SIGNOSERVER，允许它以与其他信号相同的方式进行处理。

但这种方法也有缺点。首先，不是每种语言都具有异常或信号。还有一点是不得不编写异常或信号处理器程序，这破坏了我们努力想要获得的透明性。假定你是一位程序员，你的老板要你编写一个求和的过程。你微笑着告诉她会在 5 分钟内完成编写、测试并进行文档化。然后她说你还应该编写一个异常处理程序以防万一。这时就很难再继续幻想远程过程与本地过程没有区别了，因为在单处理器系统中，为“不能定位服务器”编写异常处理器是很少见的请求。关于透明性，我们就介绍这么多。

8.3.2.2 请求消息丢失

第 2 类失败类型是请求消息丢失。这是最容易处理的一种失败：只需要使操作系统或客户存根在发送请求时开启一个定时器即可。如果在返回应答或确认之前定时器超时，那么就重新发送消息。如果消息真的丢失了，则服务器不能辨别原始的与重传的消息，但每件事情都顺利进行。当然，如果请求消息很频繁地丢失，使得客户放弃重传而错误地认为是服务器停机，那么这种情况就回到了“不能定位服务器”的问题。如果请求没有丢失，我们需要做的唯一的事就是让服务器知道它正在处理重传的消息。不幸的是，这件事并不简单，我们将在讨论应答丢失时讨论它。

8.3.2.3 服务器崩溃

第 3 个失败是服务器崩溃。服务器中事件的通常顺序如图 8.7(a)所示。请求到达、执行请求、发送应答。现在考虑图 8.7(b)。请求到达并且被执行，但是在发送应答之前服务器崩溃了。最后，看一下图 8.7(c)，请求再次到达，但是这次服务器在执行请求之前就崩溃了。

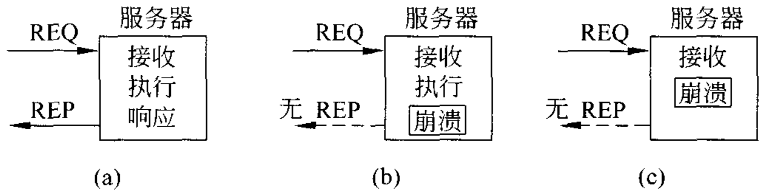


图 8.7 客户-服务器通信中的服务器

(a) 通常情况；(b) 执行之后崩溃；(c) 执行之前崩溃

图 8.7 中比较麻烦的部分在于正确对待图(b)和(c)之间的不同。在图(b)中，系统不得不向客户报告失败(也就是抛出一个异常)，而在图(c)中只是重新传输请求。问题在于客户的操作系统不能区分它们。它所知道的所有事情就是定时器超时。

关于如何处理这种情况存在三种方法(Spector 1982)。一种方法是在服务器重启之前(或重新绑定到一个新的服务器之前)等待并再次尝试操作。这种方法是在得到应答之前不断尝试，然后将应答传递给客户。这种技术被称为**至少一次语义**(at least once semantics)，它保证 PRC 至少执行一次，但是有可能执行多次。

第二种方法是立刻放弃并报告失败。这种方法被称为**最多一次语义**(at most once semantics)，它保证 RPC 最多执行一次，但是有可能一次也不执行。

第三种方法是什么都不保证。当服务器崩溃时，客户得不到有关发生了什么的任何帮助和承诺。RPC 有可能执行任意多次。这种方法的主要优点在于它很容易实现。

这些方法没有一个是很吸引人的。吸引人的方法应该具有**恰好一次的语义**(exactly once semantics)，但是一般来说没有办法可以做到这一点。想象由打印一些文本组成的远程操作，当文本被打印之后服务器向客户发送一条完成消息。假定当客户提交请求时收到一个确认消息表明请求已经被提交给服务器。服务器可以采取两种策略：或者在通知打印机进行工作之前发送完成消息，或者在文本已经被打印之后发送。

假定服务器崩溃之后又恢复了。它向所有的客户宣布它刚才崩溃了但是现在又再次运行。问题在于客户并不知道它的打印请求是否被执行了。

客户可以采取 4 种策略。首先，客户可以决定决不重发请求，这可能要冒着文本不被打印的危险。第二，它可以决定总是重发请求，但是这种方法可能导致文本被打印两次。第三，它可能只有在没有接收到打印请求已经传送到服务器的确认时才重发请求。在这种情况下，客户认为服务器在打印请求被传送到之前崩溃。第四种也是最后一种策略是只有接收到打印请求的确认时才重发请求。

由于服务器具有两种策略而客户具有 4 种，所以共有 8 种结合方法。不幸的是，没有一种是令人满意的。注意，在服务器上可能发生 3 种事件：发送完成消息(M)、打印文本(P)和崩溃(C)。这些事件可能以 6 种不同的次序发生：

(1) M→P→C：在发送完成消息和打印文本之后发生崩溃；

(2) M→C(→P)：在发送完成消息之后，打印文本之前发生崩溃；

(3) P→M→C：在发送完成消息和打印文本之后发生崩溃；

(4) P→C(→M)：首先打印文本，然后在发送完成消息之前发生崩溃；

(5) C(→P→M)：在服务器做任何事之前发生崩溃；

(6) C(→P→M)：在服务器做任何事之前发生崩溃。

括号表示由于服务器已经崩溃所以不会发生该事件。图 8.8 说明了所有可能的组合。很容易验证,没有一种客户策略和服务器策略的结合可以在所有可能的事件顺序下正确工作。根本在于客户从来都不能知道服务器是在打印文本之前还是之后发生崩溃。

客户				服务器		
	策略 M → P			策略 P → M		
重发策略	MPC	MC(P)	C(MP)	PMC	PC(M)	C(PM)
总是	DUP	OK	OK	DUP	DUP	OK
从不	OK	ZERO	ZERO	OK	OK	ZERO
仅当确认时	DUP	OK	ZERO	DUP	OK	ZERO
仅当没有确认时	OK	ZERO	OK	OK	DUP	OK

OK=文本被打印一次

DUP=文本被打印两次

ZERO=文本根本没有被打印

图 8.8 出现服务器崩溃时客户与服务器策略的不同组合

简而言之,服务器崩溃的可能性从根本上改变了 RPC 的本性,很明显地把单处理器系统与分布式系统区别开来。在前者中,服务器崩溃也暗示着客户的崩溃,所以恢复不但是不可能的也是不需要的。而在后者中有可能也有必要采取行动。

8.3.2.4 应答消息丢失

应答丢失处理起来也比较困难。直接的解决方法就是再次依赖客户的操作系统设置的定时器。如果客户在合理的时间内没有得到响应,那么就再次发送请求。这种方法的问题在于客户不能真正确认为什么没有得到响应。是请求或响应丢失还是服务器慢? 这造成了一些不同。

特别是,一些操作可以安全地重复多次而不会造成任何损害。例如请求一个文件的前 1024 个字节就没有副作用,可以执行多次而没有危害。这种请求被称为是**幂等的**(idempotent)。

现在考虑请求银行服务器从一个账户向另一个账户转账 100 万美元的请求的情况。如果请求到达并执行了,但是响应丢失了,而客户不知道这一点,它就会重发消息。银行服务器将把这个请求解释为一个新的请求,并再次执行它。这样就转账了 200 万美元。想象一下这种响应丢失 10 次吧! 传输货币不是幂等的。

解决这个问题的一种方法是按幂等的方式组织所有的请求。但是在实际中,很多请求(例如货币兑换)在本质上是非幂等的,所以就需要一些别的东西。另外一种方法是在客户为每个请求分配一个序列号。通过在服务器上跟踪从每个客户收到的最近序列号,服务器可以分辨原始的请求与重发的请求并拒绝执行第二次发出的请求。但是服务器还是要向客户发送响应。注意这种方法需要服务器维护对每个客户的管理。一个附加的防卫措施是在消息头中添加 1 位以区分是最初的请求还是重发的请求(这种思想在于执行原始请求是安全的,而执行重发请求就需要更多的小心)。

8.3.2.5 客户端崩溃

最后一种失败是客户崩溃失败。如果客户向服务器发送请求，请求做一些事情，但在服务器回复之前崩溃了，那么会发生什么呢？这时，虽然计算是活动的，但没有双亲等待结果，这种不需要的计算被称为**孤儿**(orphan)。

孤儿会引起多种问题。它们至少是在浪费 CPU 周期。它们也可以锁定文件或占用有价值的资源。最后，如果客户重新启动并再次进行 RPC，而孤儿的结果又立即返回了，那么就会造成结果的混淆。

如何处理孤儿呢？Nelson(1981)提出了4种方法。第一种方法是在客户存根(stub)发送 RPC 消息前进行日志记录来说明要做什么。日志被保留在崩溃之后仍然存在的磁盘或其他介质上。在重新启动之后，对日志进行检查然后明确地杀死孤儿。这种方法被称为**孤儿消灭**(extermination)。

这种方法的缺点在于为每个 RPC 都进行写磁盘操作要付出的昂贵代价。另外它也可能不能工作，因为孤儿本身也可以进行 RPC，这样就创建了更多的后代，因而对它们进行定位很难，甚至是不可能的。最后，网络可能由于网关故障而被分隔，这使得即使定位了它们，要杀死它们也是不可能的。总之，这不是一种有前途的方法。

在称为**再生**(reincarnation)的第二种方法中，不需要书写磁盘记录就可以解决所有这些问题。它的工作方式是把时间分为顺序编号的时期。当客户重启时，就向所有的机器广播一个消息说明一个新时期的开始。当这样的广播到达时，所有与那个客户有关的远程计算都被杀死。当然，如果网络被分隔，那么一些孤儿还能活下来。但是，当它们向回报告计算结果时，其中包含一个过期的时期号，这就使得检测到它们很容易。

第三种方法是第二种方法的变种，但是没那么严厉，它被称为**优雅再生**(gentle reincarnation)。当时期广播到达时，每台机器都进行检查来查看是否存在远程计算，如果有，那么就尝试定位它的拥有者。只有当不能找到拥有者时才杀死该计算。

最后一种方法称为**到期**(expiration)，其中每个 RPC 都被给定一个标准的时间量 T 来进行工作。如果到时不能结束，那么就必须显式地请求另外的时间量。这是很麻烦的。另一方面，如果在崩溃之后，客户在重启之前等待了时间 T，那么就可以确认抛弃所有的孤儿。这里要解决的问题是选择一个合理的 T 值以满足不同 RPC 的需要。

在实践中，这些方法都不令人满意。更糟的是，杀死一个孤儿可能带来无法预料的结果。例如，假定一个孤儿获得了一个或多个文件或数据库记录上的锁。如果突然杀死孤儿，那么这些锁就会被永远保持。而且孤儿可能已经在不同的远程队列中设置了实体以便将来某个时候启动其他进程，这样即使杀死孤儿也不能删除它的所有痕迹。在(Panzieri 和 Shrivastava 1988)中更详细地讨论了孤儿消灭问题。

8.4 可靠的组通信

如果考虑一下由复制获得的进程恢复有多重要，那么就会知道可靠的多播服务也是重要的了。多播服务保证了消息被传送给进程组中的所有成员。不幸的是，可靠多播的实现

被证明是令人惊讶的困难。在本节中，我们更深入地来探讨与把消息可靠地传送到一个进程组相关的问题。

8.4.1 基本的可靠多播方法

尽管大多数的传输层都提供了可靠的点到点通道，它们很少能提供到一组进程的可靠通信。最好的也只是让每个进程都建立到想要与之通信的每个其他进程的点到点连接。显然，这样的组织不是有效的，因为它浪费了网络带宽。不过，如果进程数目较少，通过多个可靠的点到点通道来获得可靠性是一种简单而直接的方法。

要做到这一点，我们需要准确地定义什么是可靠多播。直观地说，它意味着发送到一个进程组的消息被传递给该组中的每个成员。但是，如果在通信期间有一个进程加入该组会发生什么呢？这个进程也应该接收消息吗？同样，我们也应该确定如果一个（发送）进程在通信期间崩溃会发生什么？

要解决这样的问题，就必须区分存在故障进程时的可靠通信与假定所有进程都正确操作时的可靠通信。在前一种情况下，如果能保证所有正常的组成员都接收到消息，那么就认为多播是可靠的。其中的窍门在于除了不同的次序约束之外，还应该就在传递消息之前组看上去是什么样子达成协议。我们在下面讨论原子多播，然后再回到这个问题上来。

如果假定存在一个协议说明谁是组的成员，那么问题就会简单一些。特别是，如果我们假定进程不会失败，而且在通信进行期间不会有进程加入或离开组，那么可靠多播就简单地意味着每个消息都应该被传递到组的每个当前成员处。在最简单的情况下，不要求所有的组成员都按同样的顺序接收消息，但是有时需要这个特性。

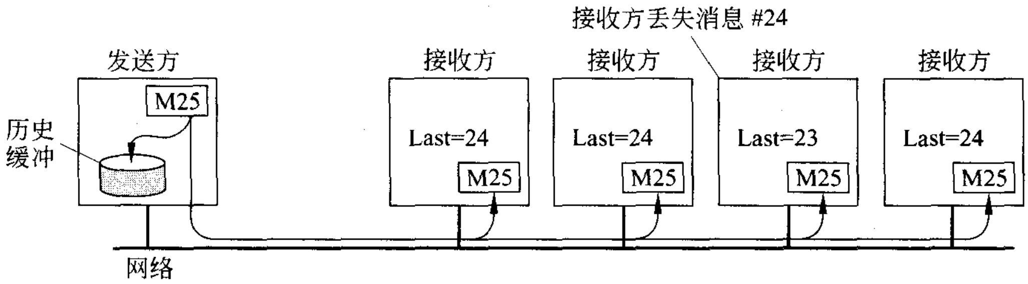
只要接收方的数目有限，那么这种较弱形式的可靠多播实现起来就相对容易。考虑一下单个发送方想要把一个消息多播发送给多个接收方的情况。假定底层的通信系统只需要不可靠的多播，那么这意味着多播消息会以某种方式丢失，它只能传送给其中一些接收方而不是全部。

一种简单的方法如图 8.9 所示。发送进程为它发送的每个多播消息分配一个序列号。假定消息按它们被发送的次序进行接收。在这种方式中，接收方很容易探测到消息丢失。每个多播消息都在发送方本地的一个历史缓存器中进行存储。假定发送方知道接收方，那么发送方就简单地在每个接收方都返回一个确认之前在历史缓存器中保留消息。如果接收方探测到它丢失了一个消息，那么就返回一个否定确认，请求发送方重发。作为选择，发送方可以在某个时间内没有接收到所有确认的情况下自动重发消息。

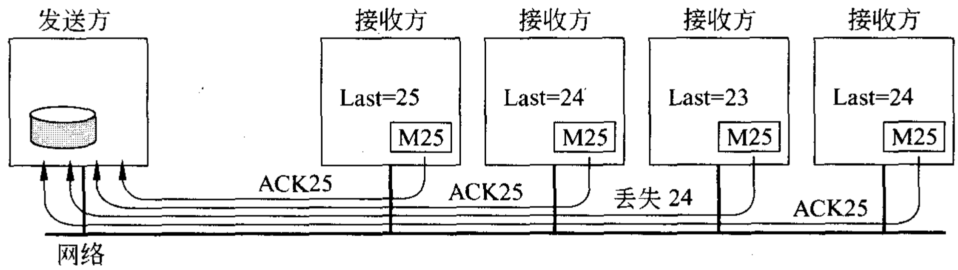
可以在多种设计中进行平衡。例如，要减少返回给发送方的消息数目，确认可能被附加在其他的消息中。重发消息也可以使用到每个请求进程的点到点通信来进行，或使用发送到所有进程的单一多播消息。要了解更多的细节，请参见（Defago 等 2004）。

8.4.2 可靠多播中的可扩展性

刚才介绍的可靠多播方法的主要问题是它不支持有很多接收方的情况。如果有 N 个接收方，那么发送方必须准备接收至少 N 个确认。如果有很多的接收方，那么发送方可能



(a)



(b)

图 8.9 当所有的接收方已知而且假定不会失败时的简单的可靠多播方法
(a) 消息传递；(b) 反馈

被大量的反馈消息淹没，这称为反馈拥塞。另外，我们也需要考虑接收方可能发布在广域网上的情况。

解决这个问题的一种方法是：接收方不对消息接收进行反馈，而是只在通知发送方消息丢失时才返回一个反馈消息。只返回否定确认通常可以减少反馈的规模（例如，请参阅 Towsley 等 1997），但是并不能保证永远不会发生反馈拥塞。

只返回否定确认在理论上还存在另外一个问题，发送方不得不永远在历史缓存器中保留消息。因为发送方永远不会知道消息是否已被传送到所有的接收方，它应该总是准备处理来自接收方的要求重发旧消息的请求。在实践中，发送方经过一段时间之后就从历史缓存器中删除消息以避免缓存器溢出。但是，删除一条消息是在冒险，因为重发的请求有可能不能得到响应。

对于可扩展的可靠多播有若干种建议，可以在（Levine 和 Garcia-Luna-Aceves 1998）中找到对不同方法的比较。我们现在只简单地讨论两种非常不同的方法，它们在现存的很多方法中是具有代表性的。

8.4.2.1 无等级的反馈控制

要获得可扩展的可靠多播方案，关键问题在于减少返回给发送方的反馈消息的数目。在一些广域网应用中流行的一种模式是**反馈抑制**（feedback suppression）。这种方法是（Floyd 等 1997）所开发的**可扩展可靠多播**（Scalable Reliable Multicasting，SRM）协议的基础，它以以下方式进行工作。

首先，在 SRM 中，接收方从来都不会对多播消息的成功传送进行确认，而是只在丢失消息时进行报告。探测消息如何丢失的问题留给应用程序处理。只有否定确认作为反馈返回。当接收方发现它丢失了一条消息时，它就向组中的其他成员多播它的反馈。

多播反馈可以使组中的其他成员抑制自己的反馈。假定多个接收方都丢失了消息 m，它们中的每一个都需要向发送方 S 返回一个否定确认以使 m 被重发。但是，如果我们假定重发总是向整个组进行多播，那么只要有一个重发请求到达 S 就足够了。

由于这个原因，一个没有接收到消息 m 的接收方 R 延迟一个随机的时间然后发送反馈消息。也就是说，在过了一段随机时间之后再请求重发。如果同时有其他对 m 的重发请求到达 R，那么 R 就抑制自己的反馈，因为它知道 m 会短期内进行重发。在理想的情况下，这种方法中只有一个反馈消息到达 S，然后 S 重发 m。这种方法如图 8.10 所示。

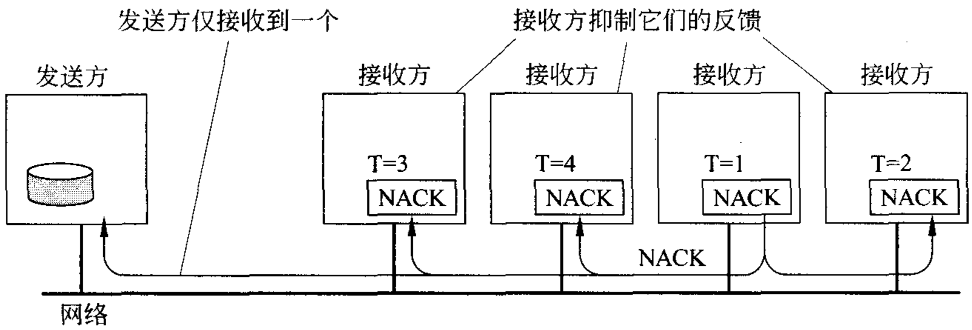


图 8.10 几个接收方要发送重发请求，但是第一个重发请求抑制了其他的请求

反馈抑制被证明具有相当好的可扩展性，它被大量的 Internet 应用程序，例如共享白板用作底层机制。但是这种方法也有一些严重的问题。首先，要确保只有一个重发请求返回给发送方需要每个接收方都对反馈消息进行相当准确的调度。否则还是会有很多接收方同时返回它们的反馈。在散布于广域网上的多个进程组中设置定时器不是容易的事。

另外一个问题是多播反馈也会中断其他成功接收消息的进程。换句话说，其他接收方不得不接收并处理对它们无用的消息。要解决这个问题，唯一的方法就是使没有接收到消息 m 的进程加入到一个独立的多播组来接收 m，在(Kasera 等 1997)中对此进行了说明。不幸的是，这样的方法要求以一种效率很高的方式来管理组，这在广域系统中很难做到。一种更好的方法是把丢失相同消息的接收方编组，共享同一个反馈消息和重发的通道。可以在(Liu 等 1998)中找到这种方法的细节。

使接收方进行本地恢复有助于提高 SRM 的可扩展性。特别是，如果一个成功地接收到消息 m 的接收方收到一个重发请求，它就可以决定甚至在重发请求到达原始的发送方之前就多播 m。可以在(Floyd 等 1997；Liu 等 1998)中找到更多的有关细节。

8.4.2.2 分等级的反馈控制

刚才介绍的反馈抑制基本上是一种无等级的方法。但是，要在非常大的接收方组中获得可扩展性，就需要采用分等级的方法。从根本上说，可靠多播的分等级方法以图 8.11 所示的方式进行工作。

为了简化，我们假定只有一个发送方需要向一个非常大的接收组进行多播。接收组分为很多子组，组织成树的形式。包含发送方的子组构成了树的根。在每个子组中，任何一种可以为小的组工作的可靠多播方案都可以使用。

每个子组都指定一个本地协作者，它负责处理子组中包含的接收方的重发请求(Hofmann 1996)。本地协作者具有自己的历史缓存器。如果协作者本身丢失了消息 m，它

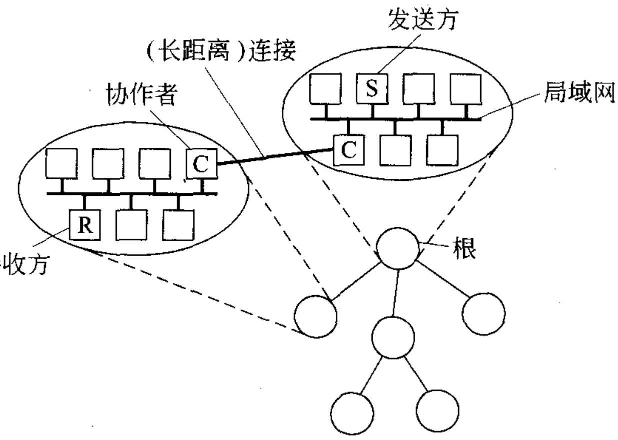


图 8.11　分等级的可靠多播。每个本地协作者都把消息转发给它的孩子然后再处理重发请求

就请求父子组的协作者重发 m。在基于确认的方法中，如果接收到消息，本地协作者就向它的父亲发送一个确认。如果协作者从它的子组的所有成员以及它的孩子那里接收到对消息 m 的确认，那么就可以从历史缓冲中删除 m。

分等级的方法中的主要问题在于树的建立。在很多情况下需要动态建立树。一种方法是使用底层网络的多播树（如果存在的话）。在原理上，这种方法强化网络层中的每个多播路由器的作用，可以按照刚才所讲的方式把它们作为本地协作者来使用。不幸的是，对现存的计算机网络进行改选并不容易。

总之，建立可以扩展到跨越广域网络的大量接收方的可靠多播是一个困难的问题。不存在一个最好的方法，每种方法都会导致新的问题出现，所以在我们被这些问题难住之前还需要在这个领域中进行大量的研究。

8.4.3　原子多播

现在我们返回到需要在存在进程失败的情况下获得可靠多播的情况。特别是，在分布式系统中经常需要保证消息要么就被发送给所有的进程，要么就不向任何进程一个也不发送。另外，如果传送的话，通常还需要所有的消息都按相同的顺序发送给所有的进程。这种方式称为**原子多播问题**（atomic multicast problem）。

为了理解为什么原子性的重要性，我们来考虑一个作为在分布式系统上建立的应用程序基础的复制数据库。分布式系统提供了可靠的多播功能。特别是，它允许建立可以可靠接收消息的进程组。因此，复制的数据库作为一组进程被建立，每个进程用于一个副本。更新操作总是向所有副本进行多播，然后在本地执行。换句话说，我们假定使用的是活动的复制协议。

假定现在要执行一系列的更新，但是在一个更新执行期间，一个副本崩溃了，于是该副本上的更新就丢失了。但是另一方面，在其他的副本上的更新被正确执行了。

当崩溃的副本恢复时，它可以恢复到它崩溃之前的状态，但是它可能会错过一些更新。在这一点上，让它与其他副本保持状态一致是基本的要求。使一个副本与其他副本保持相同的状态需要我们确切地知道错过了哪些操作以及它们是按什么次序执行的。

现在假定底层的分布式系统支持原子多播。在这种情况下，在一个副本崩溃之前发送给所有副本的更新操作或者是在所有的正常副本上都执行，或者是一个都不执行。特别是，在使用原子多播的情况下，只有在所有正确操作的副本就组成员关系达成一致时，操作才能执行。换句话说，如果其余的副本达成一致，认为崩溃的副本不再属于该组，那么更新就可以执行。

当崩溃的副本恢复时，它就被强迫再次加入该组。在它重新注册为组成员之前没有更新操作转发给它。加入一个组需要使它的状态与其他的组成员保持一致。因此，原子多播可确保没有故障的进程对数据库保持一致的视图。当一个副本从故障中恢复并重新加入组时，原子多播强制它与组的其他成员一致。

8.4.3.1 虚拟同步

存在进程失败时的可靠多播可以根据进程组和组成员关系的变化进行准确的定义。下面我们来区分一下接收消息和发送消息。特别指出的是，我们采用如图8.12中的模式，其中的分布式系统由通信层组成。消息在这个通信层中进行发送和接收。接收到的消息在发送到高层的应用程序之前放在通信层的本地缓存器中。

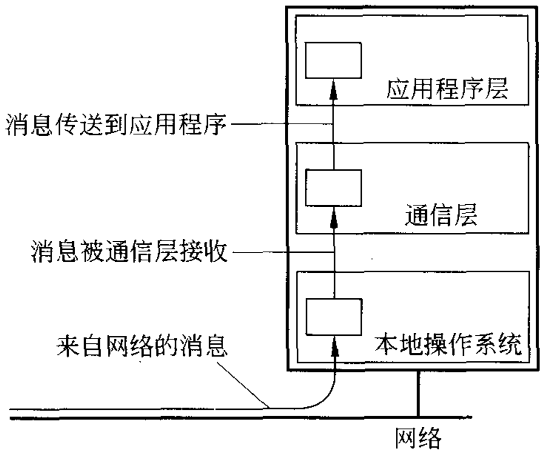


图8.12 分布式系统中的消息接收和消息发送的逻辑组织

原子多播的总体思想是多播消息 m 应该唯一地跟它应该传送到的一个进程表相关联。这个表对应于一个组视图，也就是关于包含在组中的进程集的视图。很重要的一点是列表中的每个进程都具有相同的视图。换句话说，它们都同意 m 应该被传送给它们中的每一个而且不被传送到别的进程。

现在假定 m 在发送方具有组视图 G 时进行多播。此外，假定在发生多播时另一个进程加入或离开了组。组成员关系的这个改变被宣告给 G 中的所有进程。换句话说，通过多播一个 vc 消息宣布一个进程加入或离开，视图的改变就发生了。现在有两个多播消息在同时进行传输：m 和 vc。我们需要保证要么在 vc 传送到 G 中的每个进程之前把 m 传送到每个进程，或者根本不传送 m。要注意到这个要求有些类似于第6章中讨论过的全序多播。

这里出现了一个问题，如果 m 没有被传送到任何一个进程，怎么能说这种协议是可靠的多播协议呢？原则上，只有一种情况下 m 的传送允许失败：当组成员关系的改变是由于 m 的发送方崩溃造成的结果时。在这种情况下，或者是 G 里的所有成员都知道这个新成员的异常结束，或者没有一个成员知道。作为选择，m 可以被每个成员忽略，这对应于在发送 m 之前发送方崩溃的情况。

这种更强的可靠多播保证多播到组视图 G 的消息被传送给 G 中的每个正常进程。如果消息的发送方在多播期间崩溃，那么消息或者被投递给所有剩余的进程，或者被每个进程忽略。具有这种属性的可靠多播被称为**虚拟同步**（virtually synchronous）（Birman 和 Joseph 1987）。

考虑一下图 8.13 所示的 4 个进程。在某个时刻，进程 P1 加入了一个进程组，然后这个进程组就由 P1、P2、P3 和 P4 组成。在多播了一些消息之后，进程 P3 崩溃了。但是在崩溃之前它成功地将消息多播到了进程 P2 和 P4，但是没有多播到 P1。但是，虚拟同步保证这个消息根本不会被传送，这就有效地建立起一种情况，使得在 P3 崩溃之前消息从来没有被发送。

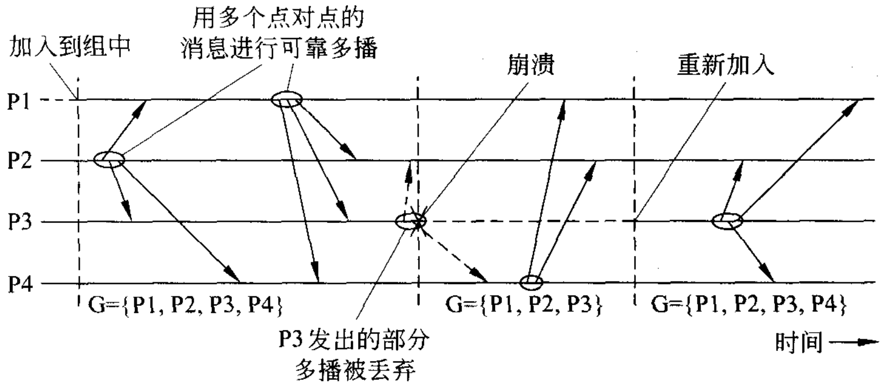


图 8.13　虚拟同步多播的原理

在从组中删除 P3 之后，通信在剩余的组成员之间进行。后来 P3 恢复了，它可以在它的状态进行更新之后再次加入该组。

虚拟同步的原理是所有的多播都在视图改变之间进行。换句话说，视图改变作为一个屏障，不能跨越它进行多播。它类似于在上一章中讨论过的分布式数据存储中的同步变量。在视图改变时进行的所有多播都在视图改变生效之前完成。虚拟同步的实现并不烦琐，下面我们进行详细的讨论。

8.4.3.2　消息排序

虚拟同步允许应用程序开发者认为多播是在不同的时期中发生的，不同的时期可以按照组成员关系的改变来划分。然而还没有谁发表关于多播的排序的论文。通常有 4 种不同的排序方法：

(1) 不排序的多播；

(2) FIFO 顺序的多播；

(3) 按因果关系排序多播；

(4) 全序多播。

可靠的、不排序多播(reliable, unordered multicast)是一种虚拟同步多播，它对接收不同进程发送的消息的次序不做任何保证。要解释这一点，我们假定可靠多播由一个提供发送和接收原语的库支持。接收操作在消息发送到它之前阻塞调用进程。

现在假定发送方 P1 把两条消息多播给一个组，同时该组中的两个其他进程正在等待消息到达，如图 8.14 所示。假定在多播期间进程不会崩溃也不会离开该组，有可能 P2 的通信层先接收到消息 m1，然后是 m2。因为没有消息顺序约束，所以消息可能是按它们被接收的顺序被传送给 P2。相反，P3 的通信层有可能首先接收到 m2，然后才是 m1，然后把它们按相同的顺序传送给 P3。

在**可靠的 FIFO 多播**(reliable FIFO-ordered multicast)情况下，通信层被强制按照消息

进程 P1	进程 P2	进程 P3
发送 m1	接收 m1	接收 m2
发送 m2	接收 m2	接收 m1

图 8.14 同一个组中的通信进程。每个进程中时间的顺序按垂直顺序排列

发送的顺序传送来自同一进程的消息。考虑如图 8.15 所示的在一个四个进程的组中的通信。使用 FIFO 的顺序,只会介意一件事情,消息 m1 总是在 m2 之前传送,同样,消息 m3 总是在 m4 之前传送。组中的所有进程都必须遵守该规则。换句话说,当 P3 的通信层先接收到 m_out2 时,在把它传送给 P3,之前必须等待 m_out1 的接收与传送。

进程 P1	进程 P2	进程 P3	进程 P4
发送 m1	接收 m1	接收 m3	发送 m3
发送 m2	接收 m3	接收 m1	发送 m4
	接收 m2	接收 m2	
	接收 m4	接收 m4	

图 8.15 同一组中的 4 个进程,具有两个不同的发送方,在 FIFO 多播下可能的消息传送顺序

但是,关于由不同进程发送的消息的传送顺序没有约束。换句话说,如果进程 P2 在 m3 之前接收到了 m1,它可能以那种顺序传送这两个消息。同时,进程 P3 可能在接收到 m1 之前接收到了 m3。FIFO 顺序说明 P3 可能在传送 m1 之前传送 m3,尽管这种传送顺序与 P2 不同。

最后,采用**可靠的按因果关系排序多播**(reliable causally-ordered multicast)来传送消息可以保留不同的消息之间的潜在因果关系。换句话说,如果在因果关系的消息 m1 在另一个消息 m2 之前,那么无论它们是否由同一个发送方多播的,每个接收方的通信层都应该在接收并传送 m1 之后传送 m2。注意按因果关系排序的多播可以使用第 5 章中讨论的向量时间戳来实现。

除了这三种排序,还有一种对消息传送的附加约束的排序被称为完全排序。**完全排序的传送**(total-ordered delivery)意味着不论消息传送是无序、FIFO 顺序还是按因果关系排序,都需要在传送消息时,对所有的组成员按照相同的次序来传送。

例如,使用 FIFO 与完全排序多播的结合,图 8.14 中的进程 P2 和 P3 都可以首先传送消息 m3,然后传送消息 m1。但是,如果 P2 在 m3 之前传送了 m1,而 P3 在 m1 之前传送了 m3,那么就可能违背全序的约束。注意也应该考虑采用 FIFO 排序。换句话说,m2 应该在 m1 之后传送,而 m4 应该在 m3 之后传送。

提供了全序的消息传送的虚拟同步可靠多播称为**原子多播**(atomic multicasting)。使用上面讨论的三种不同的消息排序约束,可以有 6 种形式的可靠多播,如图 8.16 所示(Hadzilacos, Toueg 1993)。

8.4.3.3 实现虚拟同步

现在我们来考虑虚拟同步可靠多播的实现。实现这种多播的一个例子是 Isis,一个已经实际使用了几年的容错分布式系统。我们首先来讨论一下在(Birman 等 1991)中说明的

多播	基本的消息排序	完全排序传送?
可靠多播	无	不
FIFO多播	FIFO排序传送	不
按因果关系多播	按因果关系传送	不
原子多播	无	是
FIFO原子多播	FIFO排序传送	是
按因果关系的原子多播	按因果关系传送	是

图 8.16 6种不同的虚拟同步可靠多播

有关它的实现的一些问题。

Isis中的可靠多播使用了底层网络中可用的可靠的点到点通信工具,特别是TCP。把消息m发送给一个进程组是通过把m可靠地发送给每个组成员来实现的。作为结果,尽管可以保证每个传输都成功,但还是不能保证所有的组成员都接收到m。特别是,发送方可能在把m传输给每个成员之前崩溃了。

除了使用可靠的点到点通信以外,Isis还假定来自同一个源的消息在通信层按照发送它们的顺序接收。在实践中,可以使用点到点通信的TCP连接来解决这个问题。

需要解决的主要问题是要保证发送到视图G的所有消息在组成员关系发生改变之前被传送到G中的所有正常进程。需要注意的第一个问题是确保G中的每个进程都接收到发送给G的所有消息。注意,因为将消息m发送到G的发送方可能在完成多播之前崩溃,有可能G中的进程永远不会接收到m。因为发送方已经崩溃,这些进程应该从其他的地方得到m。下面解释进程如何探测到它丢失了消息。

要解决这个问题,可以让G中的每个进程在确认G中的所有进程接收到m之前保留m。如果G中的所有进程都接收到m,那么m就被称为是**稳定的**(stable)。稳定的消息只允许传送一次。要确保稳定性,在G中选择一个任意的(可操作的)进程并要求它向其他所有进程发送m就足够了。

更明确地说,假定当前视图为G_i,但是有必要建立下一个视图G_{i+1}。为了不失一般性,我们可以假定G_i和G_{i+1}最多相差一个进程。进程P在接收到视图改变的消息时意识到视图发生改变。这样的消息可能来自想要加入或离开组的进程,或者来自G_i中被探测到失败并要被删除的进程,如图8.17(a)所示。

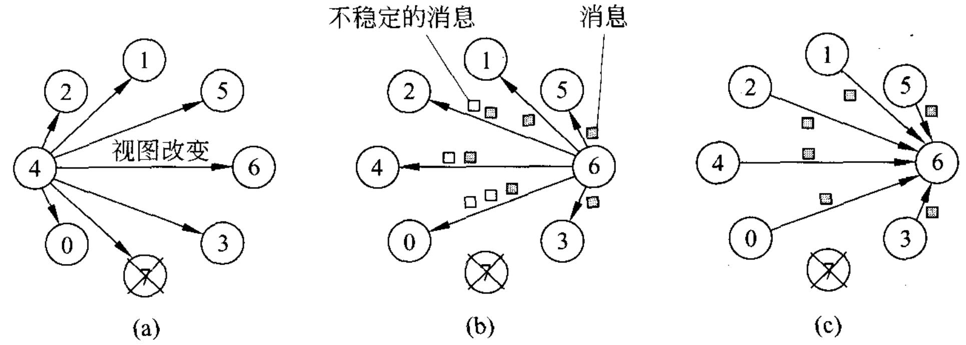


图 8.17 实现虚拟同步的过程

(a) 进程4注意到进程7已经崩溃并发送一个视图改变;

(b) 进程6发送它的所有不稳定消息,然后发送一个flush消息;

(c) 当进程6从其他每个进程那里都接收到一个flush消息时就建立了一个新的视图

当进程 P 接收到 G_{i+1} 的视图改变消息时，它首先把它拥有的 G_i 中任何不稳定消息的副本转发给 G_{i+1} 中的每个进程，然后把它标记为稳定的。回忆一下，Isis 假定点到点的通信是可靠的，这样转发的消息就不会丢失。这样的转发保证，G_i 中已经至少被一个进程接收到的所有消息可以被 G_i 中的所有正常进程接收。注意，选择单一的协作者转发不稳定消息就足够了。

为了表示进程 P 不再具有任何不稳定的消息，以及当其他进程可以建立 G_{i+1} 时它也就可以建立 G_{i+1}，它就多播一个 G_{i+1} 的**淹没消息**(flush message)，如图 8.17(b)所示。在进程 P 从其他每个进程那里都接收到 G_{i+1} 的淹没消息后，就可以安全地建立新的视图，如图 8.17(c)所示。

当进程 Q 接收到在 G_i 中发送的消息 m 而且 Q 还认为当前视图为 G_i 时，它就传送 m 而不考虑任何附加的消息排序约束。如果它已经接收到了 m，它就认为消息是复制的，然后丢弃它。

因为进程 Q 最后会接收到 G_{i+1} 的视图改变消息，它首先转发它的不稳定消息，然后通过发送一个 G_{i+1} 的 flush 消息来屏蔽别的消息。注意，由于底层通信层进行消息排序，一个进程的 flush 消息总是在接收到同一进程的不稳定消息之后进行接收。

迄今为止介绍过的协议的一个主要缺点在于它不能处理当宣布新的视图改变时的进程失败。特别是，它假定在 G_{i+1} 中的每个成员都建立起新的视图 G_{i+1} 之前，G_{i+1} 中的进程不会失败(这将导致下一个视图 G_{i+2})。这个问题可以通过在所有的进程还没有建立上一个改变时就宣布任意视图 G_{i+k} 的视图改变来解决。其中的细节留作练习。

8.5 分布式提交

前面讨论的原子多播问题是一个称为分布式提交的更一般化问题中的一个例子。分布式提交的问题涉及到要使一个操作被进程组中的每个成员都执行或一个成员都不执行。在可靠多播的情况下，操作就是消息的传送。在分布式事务中，操作可能是单个站点上事务的提交，它是整个事务的一部分。分布式提交中的其他例子及其解决办法在(Tanisch 2000)中进行了讨论。

分布式提交通常以设立协作者的方式建立。在简单的方案中，协作者通知所有涉及到的称为参与者的其他进程是否(本地)按要求执行操作。这种方法称为**单阶段提交协议**(one-phase commit protocol)。如果有一个参与者不能真正执行该操作，那么就说明它存在着明显的缺陷，因为它没有办法来通知协作者。例如，在分布式事务的情况下，不可能进行本地提交，因为这会破坏并发控制约束。

在实践中需要更复杂的方法，最常用的是两阶段提交协议，下面将对此进行详细讨论。这种协议的主要缺点是不能有效地处理协作者失败的情况。后来又研究出了三阶段提交，我们也要讨论它。

8.5.1 两阶段提交

两阶段提交协议(two-phase commit protocol, 2PC)是由 Gray (1978)提出的。为了不

失一般性，我们考虑一个分布式事务，其中有很多进程作为参与者，每个进程都运行在不同的机器上。假定没有故障发生，协议就由以下两个阶段组成，每个阶段又由两步组成（请参阅 Bernstein 等 1987）：

（1）协作者向所有的参与者发送一个 VOTE_REQUEST 消息。

（2）当参与者接收到 VOTE_REQUEST 消息时，就向协作者返回一个 VOTE_COMMIT 消息通知协作者它已经准备好本地提交事务中属于它的部分，否则就返回一个 VOTE_ABORT 消息。

（3）协作者收集来自参与者的所有选票。如果所有的参与者都表决要提交事务，那么协作者就进行提交。在这种情况下它向所有的参与者发送一个 GLOBAL_COMMIT 消息。但是，如果有一个参与者表决要取消事务，那么协作者就决定取消事务并多播一个 GLOBAL_ABORT 消息。

（4）每个提交表决的参与者都等待协作者的最后反应。如果参与者接收到一个 GLOBAL_COMMIT 消息，那么它就在本地提交事务，否则当接收到一个 GLOBAL_ABORT 消息时，就在本地取消事务。

表决阶段的第一阶段由第（1）、（2）步组成。第二阶段是决定阶段，由第（3）、（4）步组成。这 4 个步骤由图 8.18 中有限状态图来说明。

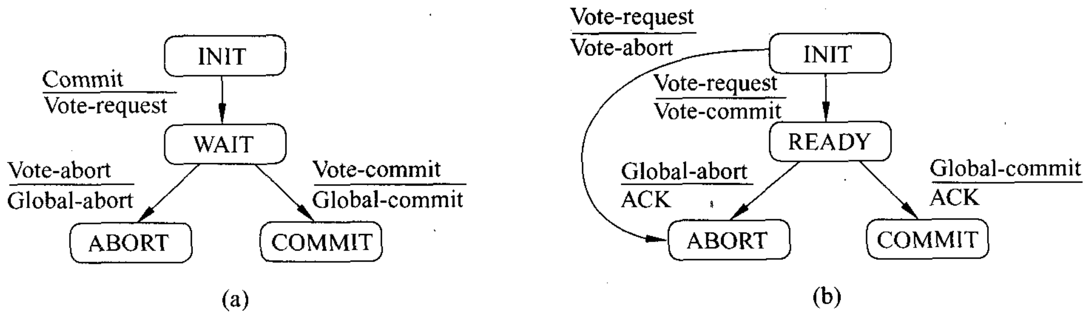


图 8.18　2PC 中协作者参与者的有限状态机

（a）2PC 中协作者的有限状态机；（b）参与者的有限状态机

当在发生故障的系统中使用这个基本的 2PC 协议时就会出现一些问题。首先，注意到协作者和参与者都具有阻塞等待消息的状态。因为当一个进程崩溃，而其他进程又正在无限等待来自该进程的消息时，这个协议很容易崩溃。因此使用了定时机制。在以下内容中将对这个机制进行解释。

看一下图 8.18 中的有限状态机，可以看出在其中三个状态中，协作者和参与者都被阻塞，等待某个消息到来。首先，参与者可能在 INIT 状态等待来自协作者的 VOTE_REQUEST 消息，如果在一段时间之后还没有接收到这个消息，那么参与者就简单地在本地中止事务，然后向协作者发送一个 VOTE_ABORT 消息。

同样，协作者可能在 WAIT 状态阻塞，等待来自每个参与者的表决。如果在某段时间之后没有接收到所有的表决，那么协作者就决定中止表决，然后向所有的参与者发送 GLOBAL_ABORT 消息。

最后，参与者可能在 READY 状态阻塞，等待协作者发送的全局表决消息。如果在给定的时间内没有接收到消息，参与者不能简单地决定中止事务。它必须查明协作者发送的

是什么消息，要解决这个问题，最简单的方法就是让每个参与者在协作者再次恢复之前阻塞。

一种更好的方法是让一个参与者P与另一个参与者Q联系，以便根据Q的当前状态来决定做什么。例如，假定Q到达了状态COMMIT，那么就有可能协作者在崩溃之前只向Q发送了GLOBAL_COMMIT消息。显然这个消息没有发送给P。因此，P应该决定进行提交。同样，如果Q的状态为ABORT，那么P也可以安全地进行中止。

现在假定Q还在状态INIT，当协作者已经向所有的参与者发送了VOTE_REQUEST消息，但是这个消息只到达P(然后它用VOTE_COMMIT消息作为应答)，而没有到达Q时就是这种情况。换句话说，协作者在多播VOTE_REQUEST时崩溃了。在这种情况下，中止事务是安全的，P和Q都可以把状态转换为ABORT。

最困难的情况发生在当Q也在READY状态等待协作者的响应的时候。特别是，如果所有的参与者都处在READY状态，那么它们就无法做出决定。问题在于尽管所有的参与者都想要提交，但是还需要协作者的表决才能做出最后的决定。因此，协议在协作者恢复之前阻塞。

在图8.19中对不同的操作进行了总结。

Q的状态	P采取的行动
COMMIT	转换到COMMIT
ABORT	转换到ABORT
INIT	转换到ABORT
READY	与其他参与者联系

图8.19 参与者P在READY状态下与另一个参与者Q联系时采取的行动

要确保进程可以真正恢复，必须在持久性存储器中保存它的状态。(如何以容错的方式保存数据在本章后面进行讨论)。例如，如果一个参与者处于INIT状态，它就可以在恢复时安全地决定在本地中止事务并通知协作者。同样，如果它已经做出了诸如处于COMMIT或ABORT状态时何时崩溃的决定时，它将按顺序恢复到该状态，并向协作者重发它的决定。

当参与者在READY状态下崩溃时就会出现问题。在这种情况下，当它恢复时不能确定自己下一步做什么，即不能确定是提交还是中止事务。因此，就要强制它与其他参与者进行联系来看下一步要做什么，这与上面介绍的在READY状态下超时的情况类似。

协作者只有两个需要跟踪的关键状态。当开始2PC协议时，应该记录它进入了WAIT状态，这样就可以在恢复之后向所有的参与者重发VOTE_REQUEST消息。同样，如果它在第二阶段中做出决定，那么只要记录该决定就足够了，这样就可以在恢复时进行重发这个决定。

在图8.20中给出了对协作者执行的操作的总结。协作者首先向所有的参与者多播一个消息以收集它们的表决。随后记录它进入了WAIT状态，然后等待来自参与者的表决。

协作者的操作：

```
write START_2PC to local log;
multicast VOTE_REQUEST to all participants;
while not all votes have been collected {
    wait for any incoming vote;
    if timeout {
       write GLOBAL_ABORT to local log;
       multicast GLOBAL_ABORT to all participants;
       exit;
    }
    record vote;
}
if all participants sent VOTE_COMMIT and coordinator votes COMMIT {
    write GLOBAL_COMMIT to local log;
    multicast GLOBAL-.COMMIT to all participants;
} else {
    write GLOBAL_ABORT to local log;
    multicast GLOBAL_ABORT to all participants;
}
```

图 8.20　在两阶段提交协议中协作者采取的操作

如果不仅没有收集到所有的表决，而且在给定的时间内没有再接收到更多的表决，那么协作者就假定有一个或多个参与者崩溃了。因此，它就中止事务，然后向(剩余的)参与者多播一个 GLOBAL_ABORT 消息。

如果没有崩溃发生，那么协作者就会收集到所有的表决。如果所有的参与者和协作者都表决要提交，那么协作者就首先把一个 GLOBAL_COMMIT 记入日志，然后发送给所有的进程。否则协作者就多播一个 GLOBAL_ABORT(在把它记入本地日志之后)。

图 8.21(a)说明了参与者执行的步骤。首先，进程等待来自协作者的表决请求。注意这个等待可以由在该进程的地址空间中运行的一个独立线程来进行。如果没有消息到来，那么事务就简单地被中止。显然协作者已经崩溃了。

参与者的操作：

```
write I NIT to local log;
wait for VOTE_REQUEST from coordinator;
if timeout {
    write VOTE_ABORT to local log;
    exit;
}
if participant votes COMMIT {
    write VOTE_COMMIT to local log;
    send VOTE_COMMIT to coordinator;
    wait for DECISION from coordinator;
    if timeout {
       multicast DECISION_REQUEST to other participants;
       wait until DECISION is received;     /* remain blocked */
       write DECISION to local log;
```

```
        }
        if DECISION==GLOBAL_COMMIT
            write GLOBAL.COMMIT to local log;
        else if DECISION==GLOBAL_ABORT
            write GLOBAL_ABORT to local log;
        } else {
            write VOTE_ABORT to local log;
            send VOTE_ABORT to coordinator;
}
```

(a)

```
actions for handling decision requests:     /* executed by separate thread */
while true {
    wait until any incoming DECISION_REQUEST is received; /* remain blocked */
    read most recently recorded STATE from the local log;
    if STATE==GLOBAL_COMMIT
        send GLOBAL.COMMIT to requesting participant;
    else if STATE==INIT or STATE===GLOBAL.ABORT
        send GLOBAL_ABORT to requesting participant;
    else
        skip;           /* participant remains blocked */
}
```

(b)

图 8.21　2PC 中参与者采取的步骤和处理输入的决定请求的步骤

(a) 2PC 中参与者争取的步骤；(b) 处理输入的决定请求的步骤

在接收到表决请求之后，参与者决定为提交一个事务进行表决，首先在本地日志中记录它的决定，然后通过发送一个 VOTE_COMMIT 消息来通知协作者。然后参与者必须等待全局决定，假定这个决定(来自协作者)按时到达，那么就简单地把它记录到本地日志中，然后执行。

但是，如果参与者在等待协作者的决定到来时超时，它就执行一个终止协议，首先向其他进程多播一个 DECISION_REQUEST 消息，然后阻塞，等待响应。当响应到达时(可能来自协作者，假定它最后恢复了)，这个参与者就把决定写入它的本地日志，然后处理它。

每个参与者都应该准备好接收来自其他参与者为做全局决定而发出的请求。假定每个参与者都开启了一个线程与参与者的主线程并发执行，如图 8.20(b)所示。这个线程在接收到决定请求之前处于阻塞状态。只有在与之相关的参与者已经做出最后决定时该线程才对其他进程有帮助。换句话说，如果协作者已经把 GLOBAL_COMMIT 或 GLOBAL_ABORT 写到了本地日志中，那么它就至少向这个进程发送了它的决定。另外，线程也要在与之相关的参与者还处在 INIT 状态时发送 GLOBAL_ABORT，就像前面讨论的那样。在所有其他的情况下，接收线程都没有帮助作用，请求的参与者也不能得到响应。

我们已经看到参与者有可能需要在协作者恢复之前阻塞。当所有的参与者都从协作者那里接收到 VOTE_REQUEST 消息并进行处理，而在同时协作者崩溃时，就会发生这样的情况。在这种情况下，参与者在要采取的最后行动的问题上不能协调一致地做出决定。因此，2PC 被称为**阻塞提交协议**(blocking commit protocol)。

有一些方法可以避免阻塞。(Babaoglu 和 Toueg 1993)描述了一种方法，它使用一个多播原语，该原语中接收方立刻把它接收到的消息向其他所有进程进行多播。这种方法允许参与者即使在协作者没有恢复时也能到达最后决定。另一种方法是三阶段提交协议，它是本章的最后一个主题，我们下面就要讨论它。

8.5.2 三阶段提交

两阶段提交的一个问题在于当协作者崩溃时，参与者不能做出最后的决定。因此参与者可能在协作者恢复之前保持阻塞。(Skeen 1981)开发了一种 2PC 的变种，称为**三阶段提交协议**(three-phase commit protocol，3PC)，它避免了在出现故障停机时的阻塞过程。尽管 3PC 在文章中广泛出现，但它在实际中用得并不多，因为在 2PC 中阻塞的情况很少发生。我们讨论这个协议，是因为它提供了解决分布式系统中的容错问题的更多看法。

像 2PC 一样，3PC 也可以用一个协作者和一些参与者来进行说明。它们各自的有限状态机如图 8.22 所示。该协议的本质在于协作者和每个参与者都满足以下两个条件：

(1) 没有一个可以直接转换到 COMMIT 或 ABORT 状态的单独状态；

(2) 没有一个这样的状态：它不能做出最后决定，而且可以从它直接转换到 COMMIT 状态。

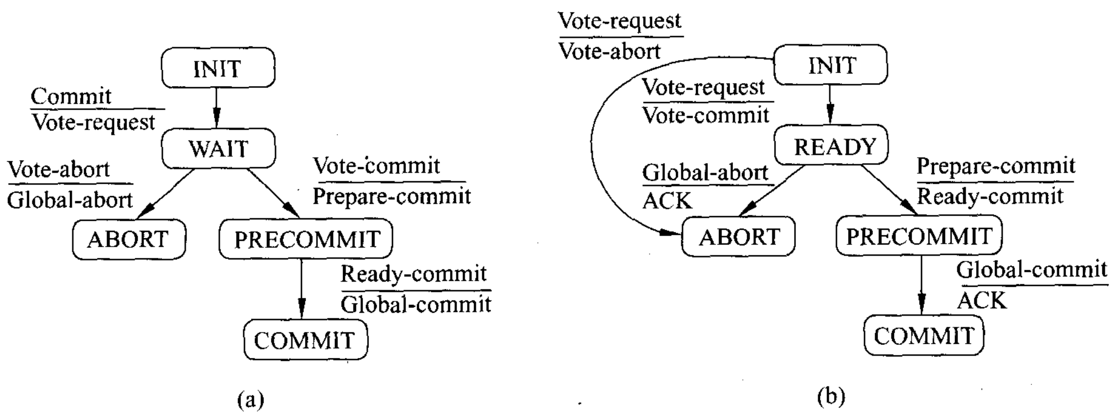


图 8.22 3PC 中协作者和参与者的有限状态机

(a) 3PC 中协作者的有限状态机；(b) 参与者的有限状态机

在(Skeen，Stonebraker 1983)中说明了这两个条件是使提交协议不阻塞的充分必要条件。

3PC 中的协作者首先向所有的参与者发送一个 VOTE_REQUEST 消息，然后等待到来的应答。如果任意一个参与者投票中止事务，那么最后决定就是中止该事务，协作者发送一个 GLOBAL_ABORT 消息。但是如果事务可以提交，就发送一个 PREPARE_COMMIT 消息。只有在每个参与者都确认它已经准备提交时，协作者才发送最后的 GLOBAL_COMMIT 消息，这时事务才真正提交。

同样会有一些进程在等待消息到来时阻塞的情况。首先，如果参与者在 INIT 状态时等待来自协作者的投票要求，假定协作者已经崩溃，那么它最终会转换到 ABORT 状态。这种情况下与 2PC 中相同。与此类似，协作者可能在 WAIT 状态等待来自参与者的投票。当超时

发生时，协作者就认为参与者已经崩溃，通过多播一个 GLOBAL_ABORT 消息来中止事务。

现在假定协作者在 PRECOMMIT 状态阻塞，当超时时它就认为一个参与者已经崩溃，但是可以知道那个参与者已经投票要提交事务。因此，协作者可以通过多播一个 GLOBAL_COMMIT 消息来指示可操作的参与者提交。另外，当崩溃的参与者再次启动时，它要依赖恢复协议来最终提交事务中属于它的部分。

参与者 P 可能在 READY 状态或 PRECOMMIT 状态阻塞。当超时时，P 只能认为是协作者崩溃，所以它需要决定下一步做什么。与 2PC 中相同，如果 P 与其他处于 COMMIT（或 ABORT）状态的参与者联系，那么 P 也应该转换到该状态。另外，如果所有的参与者都处于 PRECOMMIT 状态，那么事务就可以安全提交。

与 2PC 相同，如果另一个参与者 Q 还处于 INIT 状态，那么就可以安全地中止事务。只有别的参与者都不处于 PRECOMMIT 时 Q 才能处于 INIT 状态，记住这一点很重要。只有协作者在崩溃前已经到达 PRECOMMIT 状态并接收到每个参与者的提交表决时，参与者才可能到达 PRECOMMIT 状态。换句话说，没有参与者处于 INIT 状态而其他的参与者处于 PRECOMMIT 状态。

如果 P 可以联系的每个参与者都处于 READY 状态，那么事务应该被中止。要记住的一点是另外的参与者可能崩溃然后又恢复。但是，P（而不是任何其他可操作的参与者）知道崩溃的参与者在恢复之后的状态。如果进程恢复到 INIT 状态，那么决定中止事务是唯一正确的决定。在最坏的情况下，进程可以恢复到 PRECOMMIT 状态，但是在这种情况下，中止事务不会造成任何损害。

与 2PC 相比，3PC 的主要不同点在于以下的这种情况：崩溃的参与者可能恢复到了 COMMIT 状态而所有其他参与者还处于 READY 状态。在这种情况下，其余的可操作进程不能做出最后的决定，不得不在崩溃的进程恢复之前阻塞。在 3PC 中，只要有可操作的进程处于 READY 状态，就没有崩溃的进程可以恢复到 INIT、ABORT 或 PRECOMMIT 之外的状态。因此存活进程总是可以做出最后决定。

最后，如果进程 P 可以到达状态 PRECOMMIT（并形成多数），那么提交事务就是安全的。这里说明在所有其他的进程至少处于 READY 状态的情况下，崩溃的进程可以恢复到 READY、PRECOMMIT 或 COMMIT 状态。

有关 3PC 的更多细节可以在文献（Bernstein 等 1987）和（Chow，Johnson 1997）中找到。

8.6 恢　　复

迄今为止我们主要集中于讨论允许容错的算法。但是，一旦发生了故障，使发生故障的进程恢复到正确的状态就是基本的问题了。下面我们首先说明恢复到正确状态到底意味着什么，然后说明如何通过检查点和消息日志的方式记录并恢复分布式系统的状态。

8.6.1 概述

容错的基本要求是从错误中恢复。回忆一下，错误是系统中可能导致失败的部分。错

误恢复的总体想法是用正确的状态取代错误的状态。从本质上说，有两种形式的错误恢复。

在**回退恢复**(backward recovery)中，主要问题是要将系统从当前的错误状态回到先前的正确状态。要做到这一点，必须定时记录系统的状态，以便当发生错误时恢复到记录的状态。每次记录系统的当前状态时，就称为设置一个**检查点**(checkpoint)。

错误恢复的另一种形式是**前向恢复**(forward recovery)。在这种情况下，当系统进入错误状态时，不是回退到以前的检查点处的状态，而是尝试从可以继续执行的某点开始把系统带入一个正确的新状态。前向错误恢复机制的关键在于它必须预先知道会发生什么错误。只有在这样的情况下才有可能纠正错误并转到新的状态。

考虑一下可靠通信的实现，就很容易解释前向错误恢复与回退错误恢复之间的区别。要从分组丢失中恢复，通常的方法是让发送方重发该分组。实际上，分组重发就是尝试着回退到以前的正确状态，也就是发送丢失的分组时的状态。因此，通过分组重发获得的可靠通信就是应用回退错误恢复技术的一个例子。

还有一种可供选择的方式是使用一种称为**擦除修正**(erasure correction)的方法。这种方法是从其他成功传送的分组中建立丢失的分组。例如，在一个(n,k)块的擦除码中，一组k个源分组被编码为一组n个已编码的分组，这样任意k个已编码的分组都可以重建出原来的k个源分组。典型的值为$k=16$或$k=32$，而且$k<n\leqslant 2k$，请参阅(Rizzo 1997)。如果没有足够的分组被传送，发送方就不得不继续发送分组直到先前丢失的分组可以被重建为止。擦除修正是一种前向错误恢复方法的典型例子。

一般说来，回退错误恢复技术作为一种从分布式系统故障中恢复的机制被广泛应用。回退错误恢复的一个重要优点在于它是一种通用的方法，不依赖于任何特定的系统或进程。换句话说，它可以作为一项通用的服务集成到分布式系统的中间层中。

但是，回退错误恢复也有一些问题(Singhal, Shivaratri 1994)。首先，把一个系统或进程恢复到原先的状态通常是一个相对开销较大的操作。例如，如接下来的部分讲述的那样，要从进程崩溃或站点失败中恢复通常需要做很多工作。

第二，因为回退错误恢复机制不依赖于实际使用的分布式应用程序，不能保证一旦进行了恢复就不再发生相同或类似的失败。如果需要这样的保证，那么通常需要应用程序在恢复中参与错误处理。换句话说，回退错误恢复机制通常不能提供完全的故障透明性。

最后，尽管回退错误恢复需要设置检查点，但也决不会回退到某些状态。例如，如果一个(可能是恶意的)人从自动柜员机上取走了1000美元时，那么即使自动柜员机立即回退这些钱也很难追回。与此类似，在大多数UNIX系统中输入：

```
bin/rm -fr
```

之后，要从错误的工作目录中恢复到先前的状态绝大多数人可能是无能为力的。有些事情是不可逆的。

通过检查点可使系统恢复到先前的正确状态。但是，设置检查点通常是一个开销很大的操作，可能引起严重的性能问题。因此，很多容错的分布式系统把检查点与**消息日志**(message logging)相结合。在这种情况下，在设置检查点后，一个进程在发送消息之前对它进行记录(称为**基于发送方的日志**(sender-based logging))。一种可供选择的方法是让接收进程在把到来的消息传送到它所执行的应用程序之前先记录消息。这种方法也称为**基于接收方的日志**(receiver-based logging)。当接收进程崩溃时，需要恢复到最近的检查点的状

态，并从这个状态重放被发送的消息。把检查点与消息日志相结合使得有可能恢复到最近检查点的状态而无需付出设置检查点的代价。

在仅使用检查点方法与另外还使用日志的方法之间还有另一个重要的不同。在只使用检查点的系统中，进程将被恢复到设置检查点时的状态，它们的行为将与在发生故障之前不同。例如，因为通信时间是不确定的，消息可能以不同的次序发送，而导致接收方的不同响应。但是，如果使用了消息日志，那么就会发生从上次检查点之后事件的真实重放。这样的重放使得与外部世界相互作用更加容易。

我们来考虑一下由于用户提供了错误的输入而导致故障的情况。如果只使用检查点，那么系统为了恢复到真正相同的状态，就不得不在接受用户输入之前设置检查点。而如果还使用消息日志，就可以使用一个较老的检查点，然后在用户应该提供输入的那一点之前进行事件的重放。在实践中，使用较少的检查点与消息日志相结合的方式比使用很多检查点的方式更有效。

稳定存储

要恢复到先前的状态，就有必要安全地存储恢复所需的信息。在这种情况下，安全性意味着恢复信息不仅要经得住进程崩溃与站点故障的考验，而且还要能经得住各种存储介质故障的考验。当在分布式系统中进行恢复时，稳定存储起着重要的作用。我们在这里对它进行简单的讨论。

存储可以分为三类。首先是普通的 RAM 存储，当掉电或机器崩溃时其中的信息会丢失。然后是磁盘存储，它不受 CPU 故障的影响，但是在磁盘头损坏的情况下会丢失信息。

最后介绍一种**稳定存储**(stable storage)，它被设计可以经得住除了诸如洪水或地震之类的灾难以外的任何情况的考验。稳定存储可以使用一对普通的磁盘来实现，如图 8.23(a)所示。驱动器 2 中的每个块都是驱动器 1 中对应块的准确副本。当更新一个块时，首先对驱动器 1 中的块进行更新及验证，然后对驱动器 2 上的相同块进行相同的工作。

假定驱动器 1 已经更新，但是系统在更新驱动器 2 之前崩溃，如图 8.23(b)所示。在恢复时磁盘可以逐块进行比较。只要两个对应的块不同，就可以假定驱动器 1 是正确的(因为驱动器 1 总是在驱动器 2 之前进行更新)，所以就把新的块从驱动器 1 复制到驱动器 2。完成恢复过程之后，两个驱动器就又是相同的了。

另外一个潜在的问题是块的自然损坏。微小的尘埃或日常的磨损都可以使以前有效的块突然在没有原因或警告的情况下发生校验和错误，如图 8.23(c)所示。当探测到这样的错误时，损坏的块可以从另一个驱动器上对应的块中重新生成。

由于它的特点，稳定存储很适合于诸如原子事务这样需要高度容错的应用程序。数据被写入稳定存储，然后再读出以验证它们是否被正确写入，这样以后数据丢失的机会就变得很小。

在下面两节中，我们将讨论与检查点和消息日志有关的更多细节。(Elnozahy 等 1996)提供了分布式系统中检查点与日志的一个调查。可以在(Chow，Johnson 1997)中找到不同的算法细节。

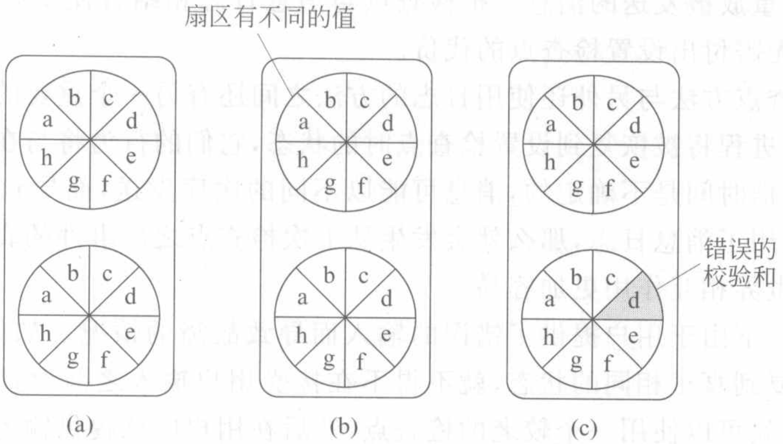


图 8.23

(a) 稳定存储；(b) 在驱动器 1 更新之后崩溃；(c) 坏点

8.6.2 检查点

在容错的分布式系统中，回退错误恢复需要系统把它的状态有规律地保存到稳定存储中。分布式系统的状态已经在第 5 章进行了讨论。特别是，我们着重强调了记录一致全局状态的必要性，它也被称为**分布式快照**（distributed snapshot）。在分布式快照中，如果进程 P 记录了一条消息的接收，那么就应该有一个进程 Q 记录了该消息的发送。毕竟消息必须是来自某个地方的。

在回退错误恢复方法中，每个进程都按时把它的状态保存到本地可用的稳定存储中。要在进程或系统失败之后进行恢复，就需要我们从这些局部状态中建立一致的全局状态。特别是，最好恢复到最近的分布式快照，这种快照也被称为**恢复线路**（recovery line）。换句话说，恢复线路就对应于最近的一致性切面，如图 8.24 所示。

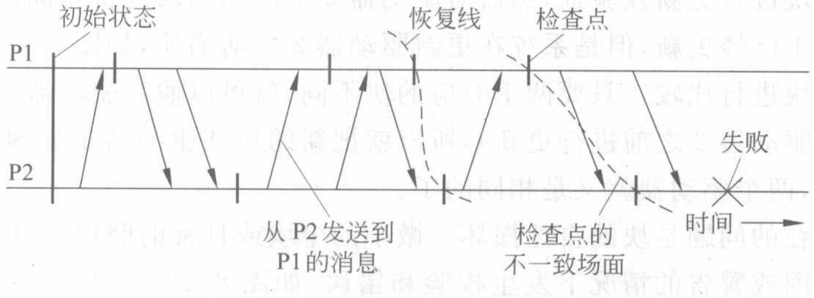


图 8.24 恢复线路

8.6.2.1 独立的检查点

不幸的是，每个进程的检查点是以一种不协调的方式来按时记录它的本地状态的，这种分布式本性使得要找到一个恢复线很困难。要找到恢复线需要每个进程都回退到它最近保存的状态。如果这些本地状态不能组成一个分布式快照，那么就需要进行进一步的回退。下面我们要说明一种寻找恢复线的方法。这种折叠回退的过程可能会导致**多米诺效应**（domino effect），如图 8.25 所示。

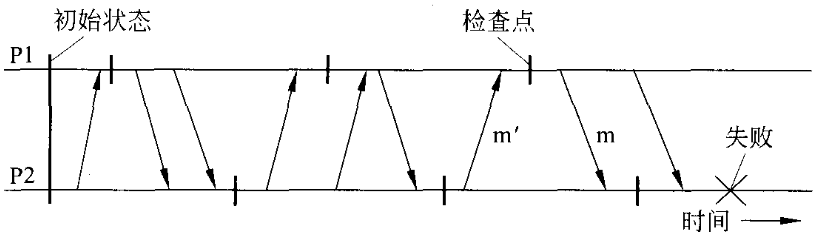


图 8.25 多米诺效应

当进程 P2 崩溃时,我们需要把它的状态恢复到最近保存的检查点。与之类似,进程 P1 也需要进行回退。不幸的是,这两个最近保存的本地状态不能组成一个一致的全局状态,P2 保存的状态表明了消息 m 的接收,但是没有其他进程说明其为发送方。因此,P2 需要回退到更早的状态。

但是,P2 要回退到的下一个状态不能用作分布式快照的一部分。在这种情况下,P1 将记录消息 m′的接收,但是没有这一消息被发送的记录事件,因此有必要把 P1 回退到以前的状态。在这个例子中,恢复线就是系统的最初状态。

由于进程独立地设置本地检查点,所以这种方法也被称为**独立的检查点**(independent checkpointing)方法。另一种可以选择的方法是像我们下面要讨论的那样设置全局协调的检查点,但是协调需要全局同步,这可能会引起一些性能问题。独立检查点的另一个缺点是每个本地存储都需要按时清理,例如通过运行一个特定的分布式垃圾收集器来进行清理。但是它主要的缺点还是恢复线的计算。

实现独立的检查点需要记录依赖关系,应按照进程可以共同回退到一个一致的全局状态的方法来进行记录。用 CPi(m)来表示进程 Pi 采用的第 m 个检查点,用 INTi(m)来表示检查点 CPi(m－1)与 CPi(m)之间的间隔。

当进程 Pi 按 INTi(m)的间隔发送消息时,它就在接收进程中加入(i,m)对。当进程 Pj 按 INTj(n)的间隔接收消息时,就使用(i,m)的索引对,它记录了 INTi(m)→INTj(n)的依赖关系。然后,当 Pj 采用检查点 CPj(n)时,它就把这个依赖关系与恢复信息的其他部分一起写入它的本地稳定存储中。

现在假定进程 Pi 在某个时刻需要回退到检查点 CPi(m－1)。要确保全局一致性,我们需要确保所有从 Pi 那里接收到了按间隔 INTi(m)发送的消息的进程都回退到在接收这个消息之前的检查点状态。特别是,在我们的例子中,进程至少需要回退到检查点 CPj(n－1)。如果 CPj(n－1)导致了不一致的全局状态,那么还需要进行更多的回退。

计算恢复线需要对每个进程在设置检查点时记录的时间间隔依赖关系进行分析。我们不讨论更多的细节,可以证明:这样的计算是相当复杂的,与协调检查点相比,独立检查点对这些计算的需求也是不尽合理的。另外,它不是支配性能因素的进程之间的协调,相反,需要把状态保存到本地稳定存储中,它还造成一些额外开销(Einozahy 等 1992)。因此,协调检查点使用得更为普遍,它也比独立检查点更为简单。

8.6.2.2 协调检查点

顾名思义,在**协调检查点**(coordinated checkpointing)中,所有的进程都同步地把它们的状态写到本地稳定存储中。协调检查点的主要优点是所保存的状态自动保持全局一致,

这样就可以避免导致多米诺效应的折叠回退。可以使用第5章中讨论的分布式快照算法来协调检查点。这种算法是非阻塞的检查点协调的一个例子。

一种比较简单的方法是使用两阶段的阻塞协议。首先协作者向所有进程多播一个CHECKPOINT_REQUEST消息。当进程接收到这样的消息时就设置一个本地检查点，通过它正在执行的应用程序将随后传递给它的消息进行排队，然后向协作者确认它已经设置了检查点。当协作者接收到所有进程的确认时，它就多播一个CHECKPOINT_DONE消息让（阻塞的）进程继续执行。

很容易看出这种方法可以使全局状态一致，因为没有即将进来的消息注册为检查点的一部分。其原因在于任何跟随在设置检查点的请求之后的消息都不被认为是本地检查点的一部分。同时，在接收到CHECKPOINT_DONE消息之前对即将发出的消息（当通过运行的应用程序传递给检查点进程的时候）在本地进行排队。

对这种算法的一种改进是只对那些依赖协作者的恢复的进程多播检查点请求而忽略其他进程。如果一个进程在设置了上个检查点之后接收到了与协作者发送的消息直接或间接相关的消息，那么这个进程就是依赖于协作者的。这称为**增量快照**（incremental snapshot）。

要设置一个增量快照，协作者只对那些在上次设置检查点之后对之发送了消息的进程多播检查点请求。当进程P接收到这样的请求时，它就把请求转发给那些在上次检查点之后P本身向之发送了消息的所有进程，等等。一个进程只转发一次请求。当确认所有的进程之后，使用第二次多播来引发实际的检查点，然后所有的进程继续执行。

8.6.3 消息日志

考虑到设置检查点是一个开销很大的操作，特别是该操作涉及到向稳定存储中写入状态，需要寻找一种可以减少检查点的数目但是还允许恢复的技术。在分布式系统中有一种重要的技术消息日志。

消息日志的基本思想是：如果消息的传输可以重放，那么我们就能够到达一个全局一致的状态而不需要从稳定存储中恢复该状态。它使用一个检查点状态作为开始点，然后简单地把从该点以后发送的所有消息都重发并进行处理。

假定采用**分段确定模式**（piecewise deterministic model），这种方法可以很好地工作。在这种模式中，假定每个进程的执行都有一连串的间隔，事件是在这些间隔中发生的。这些事件与我们在第6章中讨论过的Lamport的先发生关系中的事件相同。例如，一个事件可能是一条指令的执行、一条消息的发送等。假定分段确定模式中的每个间隔都以一个非确定性的事件，例如一条消息的接收开始。但是，这个时刻之后进程的执行就是完全确定的。一个时间间隔以发生非确定性的事件之前的最后一个事件结束。

实际上，如果以相同的非确定性事件作为开始进行重放，那么一个间隔的重放可以具有已知的结果，也就是说，以一种完全确定的方式进行重放。因此，如果我们在这样的模式中记录下所有的非确定性事件，就有可能以一种确定的方式完全重放整个进程的执行。

考虑到要从进程崩溃中恢复到一个全局一致的状态时消息日志是必需的，准确地知道什么时候对消息进行记录是很重要的。Alvisi和Marzullo(1998)介绍的方法说明，如果我

们关注现存的消息日志方案如何处理孤儿进程，那么可以容易地描述消息日志方案的特征。

孤儿进程(orphan process)就是在其他进程崩溃之后还存在的进程，但是它的状态与恢复之后的崩溃进程不一致。图 8.26 中的情况就是一个例子。进程 Q 从进程 P 与 R 那里分别接收到消息 m1 和 m2，然后把消息 m2 发送给 R。但是，与其他消息不同，消息 m2 没有进行日志记录。如果进程 Q 崩溃然后恢复，那么要恢复 Q 只需要重放记录了的消息，在我们的例子中就是 m1，因为 m2 没有进行记录，它的传输也就不会被重放，这意味着 m2 的传输不会发生。

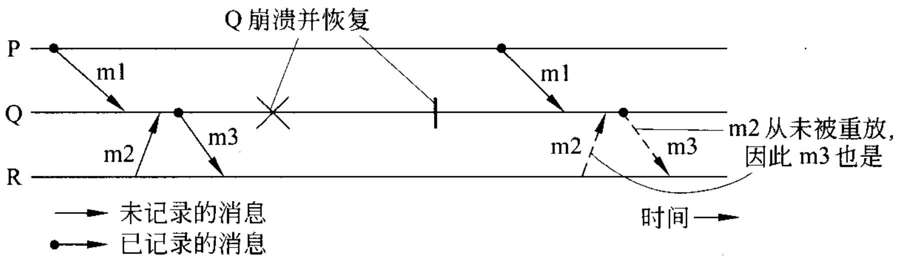


图 8.26 在恢复之后不正确的重放导致孤儿进程

但是 Q 恢复之后的情况是与它恢复之前不一致的。特别是，R 保留了在崩溃之前发送的消息，但是在重放崩溃之前发生的事情时该消息的接收与传送还没有发生。很明显，这样的不一致应该避免。

描述消息日志方案的特征

为了描述不同的消息日志方案的特征，我们使用在文献(Alvisi，Marzullo 1998)中说明的方法。认为每个消息 m 都具有头部，头部中包含了重发 m 并正确处理它所需的所有信息。例如，每个头部都包含了发送方和接收方的标识符以及一个用来识别它的序列号。另外还有一个传送号码用来确定它应该在何时被传送给接收应用程序。

如果消息不再被丢失就被称为是**稳定的**(stable)，例如因为它被写入了稳定存储。因此可以使用重放稳定消息的传输来进行恢复。

每个消息 m 都导致产生一个进程组 DEP(m)，它依赖于 m 的传送。特别是，DEP(m)由那些接收 m 的进程组成。另外，如果另一个消息 m′由于因果关系也要依赖 m 的传送，而且 m′被传送给进程 Q，那么 Q 也包含在 DEP(m)中。注意，m′是按因果关系依赖于 m 的传送的，如果它由先前传送 m 的同一个进程发送，或者它传送了另一个按因果关系依赖于 m 的传送的消息。

COPY(m)进程集由这样一些进程组成：它们具有 m 的副本，但是该副本不在它的本地稳定存储中。当一个进程传送消息 m 时，它就也成为 COPY(m)的成员。注意，COPY(m)由那些可以移交 m 的副本的进程组成，这些副本可以用来重放 m 的传输。如果所有这些进程都崩溃了，那么重放 m 的传输显然就是不可能的。

使用这些表示，现在很容易就可以准确定义孤儿进程是什么了。假定一些进程在分布式系统中崩溃了，Q 是幸存的进程之一。如果 Q 包含在 DEP(m)中，在 COPY(m)中的每个进程都崩溃了的同时存在消息 m，那么 Q 就是一个孤儿进程。换句话说，如果一个进程依赖于 m，但是没有办法来重放 m 的传输时它就是一个孤儿进程。

要避免产生孤儿进程，需要确认：如果 COPY(m)中的每个进程都崩溃了，在 DEP(m)中没有幸存的进程。换句话说，DEP(m)中的所有进程都应该已经崩溃了。如果我们能保证不论何时一个进程成为了 DEP(m)的成员，它也要成为 COPY(m)的成员，那么就可以满足这个条件。换句话说，不论何时一个进程要依赖于 m 的传送，那么它就总要保持 m 的一份副本。

可以使用两种方法来做到这一点。第一种方法称为**悲观日志协议**(pessimistic logging protocol)。这个协议认为对于每个非稳定的消息 m，都最多有一个进程依赖它。换句话说，悲观日志协议确保每个非稳定的消息 m 都最多被传送给一个进程。注意只要 m 被传送给进程 P，P 就成为 COPY(m)的成员。

最坏的情况就是 P 在 m 还没有被记录的时候就崩溃了。使用悲观日志，在传送 m 之后，如果还没有确保 m 被写到稳定存储就不允许 P 发送任何消息。因此，没有其他进程会依赖 m 的传送，这就没有了重放 m 的传输的可能性。这种方法可以避免孤儿进程。

相反，在**乐观日志协议**(optimistic logging protocol)中，在发生崩溃之后才进行实际的工作。特别是，假定对某个消息 m，COPY(m)中的每个进程都崩溃了。在乐观的方法中，DEP(m)中的任何孤儿进程都被回退到一个不再属于 DEP(m)的状态。很明显，乐观日志协议需要跟踪依赖关系，这使它们的实现变得更复杂。

像在(Einozahy 等 2002)中指出的那样，悲观日志要比乐观的方法简单得多，所以在实际的分布式系统设计中它是消息日志的首选方法。

8.6.4 面向恢复的计算

处理恢复的一个有关方法是从头开始。这种掩盖故障的方法的基本原理是，为了恢复而优化可能会更划算，从而使得系统避免处于长时间的故障之中。这种方法由称为**面向恢复的计算**(recovery-oriented computing)(Candea 等 2004a).

面向恢复的计算有几个优点。其中一个是，只需要重新启动部分系统和因特网服务器(Candea 等 2004b,2006)。为了使得可以只启动部分系统，其关键是故障被正确地局部化了。此时，重新启动只是意味着删除掉确定组件的实例以及在这些组件上进行的线程，并(经常)重新开始有关的请求。注意，故障局部化本身可能就不是一件简单的事情(Steinder 和 Sethi,2004)。

为了使重新启动成为一个确实可行的恢复技术，要求组件尽量分离，使得不同组件之间尽量少或没有关联。如果组件之间存在很强的关联，那么故障本地化和分析仍然要求在故障发生时，需要应用传统的恢复技术使这个服务器重新启动，而且这不我们上面刚讨论的技术更有效。

面向恢复的计算另一个优点是，在已变化的环境下，可以应用检查点和恢复技术而不影响系统的继续运行。这里基本思想是，如果给予程序更多的缓冲空间、内存在分配之前进行清零、改编消息传送的顺序(只要这不影响消息的语义就可以)等等，很多故障是可以避免的(Qin 等 2005)。关键思想是处理软件故障(到目前为止所讨论的很多技术针对或是基于硬件故障的)。因为软件运行是高度可确定的，改变运行环境就可以扭转局面，而且无需修理任何东西。

本章小结

容错是分布式系统设计中的一个重要主题。容错是系统的这样一种特性：系统可以掩饰故障的发生并从故障中恢复过来。换句话说，如果系统可以在发生故障时继续操作，那么它就具有容错性。

存在几种类型的故障。如果只是简单地停止了进程，那么就发生崩溃性故障。如果进程不对到来的请求做出响应，那么就发生了遗漏性故障。当进程对请求的响应过快或过慢时就发生定时性故障。如果以错误的方式响应到来的请求，那么就是响应性故障。最难处理的失败是系统出现的任意类型故障，这时称为任意性故障或拜占庭故障。

冗余是获得容错性所需的关键技术。对于进程来说，进程组的概念很重要。一个进程组由很多进程组成，它们紧密协作来提供服务。在容错的进程组中，一个或多个进程的失效不会影响到该组实现的服务的可用性。通常，组内的通信必须高度可靠，而且为了获得容错性，应该具有严格的次序和原子属性。

可靠的组通信也称为可靠多播，它具有多种不同的形式。只要组相对较小，那么实现可靠性就是可行的。但是，如果需要支持非常大的组，可靠多播的可扩展性就成为一个问题。获得可扩展性的关键问题在于减少接收方发送的反馈消息数量，这些消息用来报告多播消息(不)成功接收。

如果需要提供原子性，那么事情就变得更复杂。在原子多播协议中，关键在于每个组成员都对多播消息应该被传送给哪个成员具有相同的视图。原子多播可以根据一种虚拟同步的执行模式进行准确的定义。这种模式引入了边界的概念，在边界之间组成员关系不发生变化，而且消息也可靠地进行传输。一条消息不会跨越边界。

组成员关系改变是每个进程都需要在组成员表上达成一致的一个例子。可以使用提交协议来达成这样的协议，其中两阶段提交协议是使用最广泛的。在两阶段提交协议中，首先一个协作者检查是否所有的进程都同意执行相同的操作(也就是它们是否都同意提交)，然后在第二阶段中多播表决结果。可以使用三阶段提交协议来处理协作者崩溃的情况，而不必在协作者恢复之前使所有的进程阻塞以达成协议。

容错系统中的恢复可以通过有规律地对系统状态设置检查点来获得。检查点是完全分布式的。不幸的是，设置检查点是一个开销很大的操作。为了提高性能，很多分布式系统在检查点中结合使用了消息日志。通过记录进程之间的通信，有可能在发生崩溃之后重放系统的执行过程。

习题

1. 可靠系统通常需要提供高度的安全性，为什么？
2. 是什么使得在崩溃性故障的情况下难以实现故障停机模式？
3. 考虑一个返回过期的缓存页而不是服务器上最近更新版本的 Web 浏览器的情况，这种故障属于哪种故障？
4. 文中介绍的三倍模块冗余是否能处理拜占庭故障？
5. 图 8.2 可以处理多少失效的元素(设备加上表决者)？给出一个例子说明它能掩盖的最坏情况。

6. TMR 是否能推广到每组 5 个元素而不是 3 个？如果可以，它具有什么属性？
7. 对以下每个应用程序，你认为最多一次语义和最少一次语义哪个最好？试讨论一下。
   (a) 从文件服务器上读写文件；
   (b) 编译一个程序；
   (c) 远程银行。
8. 使用异步 RPC，客户在它的请求被服务器接受之前阻塞。失效会在什么程度上影响异步 RPC 的语义？
9. 给出一个组通信不需要消息排序的例子。
10. 在可靠多播中，通信层是否总是需要为了重发的目的而保留消息的一个副本？
11. 原子多播的可扩展性有多重要？
12. 在本章中，我们建议在一个进程集上执行更新时，原子多播可以节省时间。我们可以保证每个更新在什么程度上真正执行？
13. 虚拟同步类似于分布式数据存储中的弱一致性，组视图的改变可以作为同步点。在这种情况下，什么类似于强一致性呢？
14. 在图 8.15 的 FIFO 与完全排序相结合的多播情况下，什么是可允许的传送顺序？
15. 在虚拟同步的情况下，为协议建立下一个视图 G_{i+1} 以使它可以容忍进程发生故障。
16. 在两阶段提交协议中，为什么即使在参与者们选择一个新的协作者的情况下也不会完全消除阻塞？
17. 在我们对三阶段提交的解释中，看上去对事务的提交是基于多数表决的，是这样吗？
18. 在分段确定执行模式中，只记录消息就足够了吗，或者说，我们还需要记录其他的事件吗？
19. 解释一下如何使用分布式事务中的先写日志来从故障中恢复。
20. 无状态服务器需要设置检查点吗？
21. 通常认为，基于接收方的消息日志比基于发送方的日志更好，为什么？

第9章

安全性

我们要讨论的建立分布式系统的最后一个原则是安全性，但这绝不意味着安全性是最不重要的原则。然而，有人可以证明安全性是最难以处理的原则，因为安全需要普遍深入到整个系统中。有关安全的一个单纯的设计缺陷可能导致所有的安全措施无效。在本章中，我们集中讨论分布式系统中普遍加入的、支持安全的不同机制。

我们从介绍安全方面的基本问题开始。在对一个系统建立所有类型的安全机制并不都是真正有意义的，除非了解那些机制的使用方式以及该机制所预防的问题。这就需要我们了解要执行的安全策略。首先来讨论安全策略的概念以及有助于执行这样的策略的一些普通的机制设计问题。我们还简要地涉及到必要的加密方法。

分布式系统中的安全可以粗略地划分为两部分。一部分涉及可能处于不同机器上的用户或进程之间的通信。确保安全通信的主要机制是安全通道机制。安全通道，更明确地说，是身份认证、消息完整性以及机密性，这些内容将分别在一个单独的小节中讨论。

另一部分涉及授权，用以确保进程只获得授予它的对分布式系统内资源的访问权限。授权将在一个涉及访问控制的单独小节中介绍。除传统访问控制机制之外，我们还将集中讨论必须处理诸如代理一类的移动代码时的访问控制。

安全通道和访问控制需要分发加密密钥的机制，还需要向系统中添加或从系统中删除用户的机制。这些主题包括在称为安全管理的内容中。在一个单独的小节中，我们将讨论这样一些问题：加密密钥管理、安全组管理以及分发证明所有者有权对指定资源进行访问的证书。

9.1 安全性概述

我们以一些一般的安全性问题来开始对分布式系统中安全性的描述。首先，有必要对安全系统进行定义。我们将安全策略与安全机制区别开来，并考察明确制定了安全策略的Globus广域系统。其次，我们关注的是要为安全系统考虑一些一般的设计问题。最后，我们简要地讨论一些加密算法，这些算法在安全协议的设计中扮演关键的角色。

9.1.1 安全威胁、策略和机制

计算机系统中的安全性与可靠性的概念密切相关。非正式地说，一个可靠的计算机系

统是一个我们可以有理由信任来交付其服务的系统(Laprie 1995)。如第7章中所提到的,可靠性包括可用性、可信赖性、安全性和可维护性。然而,如果我们要信任一个计算机系统,就还应该考虑机密性和完整性。**机密性**(confidentiality)是指计算机系统的一种属性,系统凭借此属性使得信息只向授权用户公开。**完整性**(integrity)是指对系统资源的变更只能以授权方式进行。换句话说,在安全的计算机系统中,不适当的变更应该是可以察觉的,并且是可以恢复的。任何计算机系统的主要资源都是其硬件、软件和数据。

考虑计算机系统中安全的另一种角度是我们试图保护该系统所提供的服务和数据不受到**安全威胁**(security threat)。要考虑4种类型的安全威胁(Pfleeger 2003):

(1) 窃听;

(2) 中断;

(3) 修改;

(4) 伪造。

窃听(interception)是指一个未经授权的用户获得了对一项服务或数据的访问的情况。窃听的一个典型事例是两人之间的通信被其他人偷听到。窃听还发生在非法复制数据时,例如,发生在入侵者闯入文件系统中某个人的私人目录后。

中断(interruption)的一个实例是文件被损坏或丢失时所出现的情况。一般来说,中断是指服务或数据变得难以获得、不能使用、被破坏等情况。在这个意义上说,服务拒绝攻击是一种安全威胁,它归为中断类,一些人正是通过它恶意地试图使其他人不能访问服务。

修改(modification)包括对数据未经授权的改变或篡改一项服务以使其不再遵循其原始规范。修改的实例包括窃听然后改变传输的数据、篡改数据库条目以及改变一个程序使其秘密记录其用户的活动。

伪造(fabrication)是指产生通常不存在的附加数据或活动的情况。例如,一个入侵者可能尝试向密码文件或数据库中添加一项。同样,有时可能通过重放先前发送的消息来侵入一个系统。我们稍后将在本章中讨论这样的例子。

注意,中断、修改和伪造中的每一个都可视为数据篡改的一种形式。

仅仅声明系统应该能够保护其自身免受任何可能的安全威胁并不是实际建立一个安全系统的方式。首先需要的是安全需求的一个描述,也就是说,一个安全策略。**安全策略**(security policy)准确地描述系统中的实体能够采取的行为以及禁止采取的行为。实体包括用户、服务、数据、机器等。制定了安全策略之后,就可能集中考虑**安全机制**(security mechanism),策略通过该机制来实施。重要的安全机制包括:

(1) 加密;

(2) 身份认证;

(3) 授权;

(4) 审计。

加密(encryption)是计算机安全的基础。加密将数据转换为一些攻击者不能理解的形式。换句话说,加密提供了一种实现机密性的方式。另外,加密使我们能够检查数据是否被修改,从而还提供对完整性检查的支持。

身份认证(authentication)用于检验用户、客户、服务器等所声明的身份。对客户来说,基本前提是在客户可以使用一项服务之前,该服务必须了解客户的身份。通常,依靠密码对

用户进行身份认证，但对客户进行身份认证还有许多其他方式。

对一个客户进行身份认证之后，有必要检查是否授予该客户执行所请求操作的权限。对医学数据库中记录的访问是一个典型实例。取决于访问该数据库的不同人员，可能分别赋予其读取、修改记录中特定字段，以及添加或删除记录的许可。

审计(auditing)工具用于追踪各个客户的访问内容以及访问方式。虽然审计并不真正提供任何防止安全威胁的保护，但审计记录对安全破坏的分析以及随后采取措施防止入侵非常有用。出于这种原因，攻击者通常不希望留下任何可能最终导致其身份暴露的痕迹。在这种意义上，对访问记录进行攻击有时成为更冒险的事情。

示例：Globus 安全体系结构

安全策略的概念和安全机制在分布式系统中为实施这样的策略所扮演的角色通常可以通过一个具体示例给予最好的解释。我们来考虑一下为 Globus 广域系统所定义的安全策略(Chervenak 等 2000)。Globus 是一个支持大规模分布式计算的系统，其中同时使用许多主机、文件和其他资源来进行计算。这样的环境又称为计算网格(Foster 和 Kesselman 2003)。这些网格中的资源通常位于不同的管理域，这些域可以位于世界的不同地方。

因为用户和资源在数量上是巨大的，而且广泛地散布在不同管理域中，所以安全是最基本的要求。要设计并正确使用安全机制，就需要理解确实需要保护的对象，以及有关安全的假定内容。稍作简化，Globus 的安全策略必须具有下述 8 条声明，我们将在下面进行解释(Foster 等 1998)：

(1) 该环境由多个管理域组成；

(2) 局部操作(即仅在单个域内执行的操作)仅受局部网域安全策略的支配；

(3) 全局操作(即包括若干域的操作)需要执行该操作的每个域都知道发起者；

(4) 不同域中的实体之间的操作需要相互的身份认证；

(5) 用全局身份认证取代局部身份认证；

(6) 资源的访问控制仅受局部安全支配；

(7) 用户可以将权限委派给进程；

(8) 同一域内的一组进程可以共享凭证。

Globus 假设该环境由多个管理域组成，其中每个域都具有其自己的局部安全策略。假设仅因为该域在 Globus 内从而不能改变局部策略，而且 Globus 的全面策略不能覆盖局部安全决策。这样，Globus 中的安全就将自身限制于影响多个域的操作。

与这一问题相关的是，Globus 假设对一个域完全局部化的操作仅受该域的安全策略的支配。换句话说，如果一个操作是在单个域内开始并执行的，那么所有安全问题都将仅使用局部安全措施来执行。Globus 不会利用其他措施。

Globus 安全策略规定对操作的请求可以全局地也可以局部地开始。该发起者，不管是用户还是代表用户的进程，在执行该操作的每个域内都必须被该局部所了解。例如，一个用户可能具有一个映射为指定域的局部名称的全局名称。该映射发生的确切程度留待每个域自己确定。

一个重要的策略声明是不同域中实体间的操作需要相互的身份认证。这就意味着，如果一个域内的一个用户使用来自另一个域的一项服务，那么就必须检验该用户的身份。同样重要的是，必须确保用户在使用其认为正在使用的服务。在本章的后面，我们将详细讨论

身份认证。

我们将上面的两个策略问题与下面的安全需求结合在一起。如果已经检验过一个用户的身份，并且该用户已在一个域内被局部了解，那么该用户就可以像在该局部域内已经通过身份认证那样行动。这意味着，Globus 需要其系统范围的身份认证方法足够充分，从而认为用户在访问一个远程域（在这里用户是已知的）时已经通过该域的身份认证。由该域进行的其他身份认证应该是必要的。

一旦用户（或代表用户的进程）通过身份认证后，仍然需要检验与资源相关的确切访问权限。例如，希望修改一个文件的用户首先必须通过身份认证，身份认证后就可以检查该用户是否确实被允许修改该文件。Globus 安全策略声明这种访问控制决策是在所访问的资源所处的域内完全局部化决定的。

要解释第 7 条声明，让我们来考虑 Globus 内的一个移动代理程序，该代理程序通过逐个开始不同网域内的若干个操作来执行任务。这样的一个代理程序可能要花费很长时间完成其任务。为了避免与该代理程序所代表的用户进行通信，Globus 要求能够将用户权限的一个子集委派给进程。因此，通过对一个代理程序进行身份认证，然后检查其权限，Globus 应该能够允许代理程序开始一项操作而不必联系其所有者。

作为最后的策略声明，Globus 要求在单个域内运行并代表同一用户的进程组可以共享一个单独的凭证集。正如将在下面解释的那样，身份认证需要认证。这一声明不要求每个进程都具有其特有的唯一凭证集，从而在本质上开辟了身份认证的可扩展解决方案的道路。

Globus 安全策略允许其设计者集中开发一个全面的安全解决方案。通过假设每个域执行其自身的安全策略，Globus 仅注重涉及多个网域的安全威胁。特别是，该安全策略指出重要的设计问题是远程域内用户的表示方法，以及向用户或其代表分配来自一个远程网域中的资源。因此 Globus 首先需要的是跨网域的身份认证机制并使用户在远程网域内为人所知。

出于这一目的，该策略引入了两种类型的代理。**用户代理**（user proxy）是一个进程，系统允许该进程在有限的一段时间内代表用户进行操作。资源由资源代理表示。**资源代理**（resource proxy）是在指定域内运行的一个进程，用于将一个资源上的全局操作转换为遵循特定网域的安全策略的局部操作。例如，需要访问一个资源时，用户代理通常会与该资源代理进行通信。

Globus 安全结构从本质上由诸如用户、用户代理、资源代理和一般进程这样的实体组成。这些实体位于域中，并互相影响。特别是，该安全结构定义了 4 个不同的协议，如图 9.1 所示（Foster 等 1998）。

第 1 个协议准确描述了用户创建用户代理和向该代理委派权限的方法。特别是，为了使该用户代理代表该用户行动，用户要赋予该代理一个适当的认证集。

第 2 个协议指定了用户代理对远程域中的资源提出分配请求的方式。从本质上说，该协议在进行相互身份认证后通知资源代理在远程域中创建一个进程。该进程代表该用户（正如用户代理所做的），但在与被请求的资源所在的相同域内进行操作。有些资源由局限于该域的访问控制决策控制，系统赋予该进程访问这些资源的权限。

在远程域内创建的进程可以启动其他域内的附加计算。因此，在进程而不是用户代理请求远程域中的资源时就需要一个协议来分配该资源。在 Globus 中，这种类型的分配经由

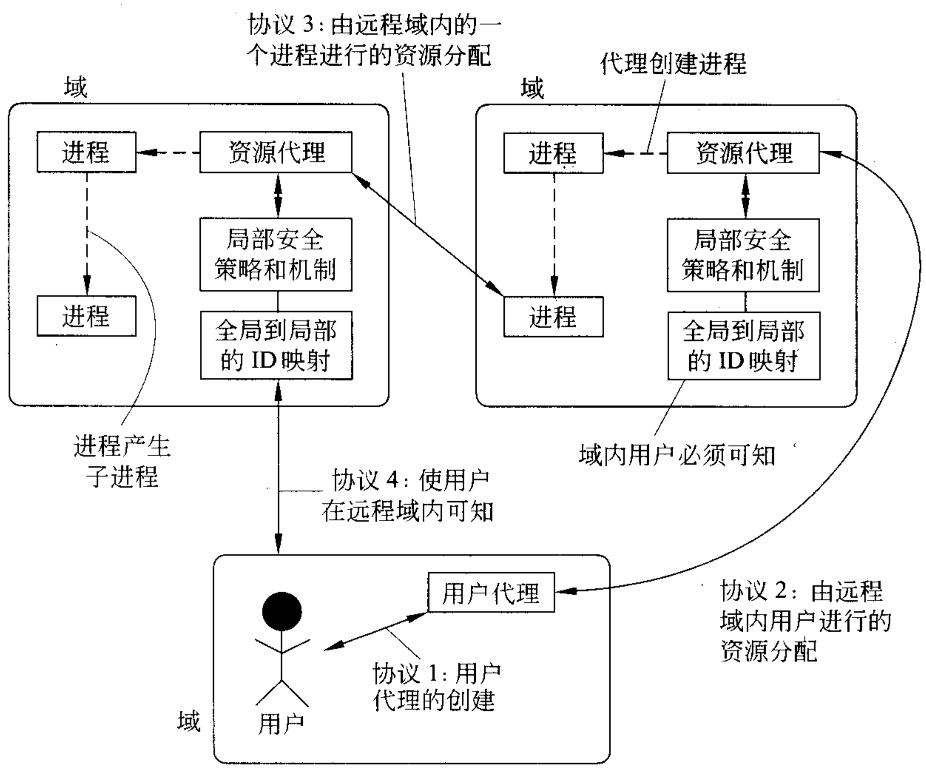


图 9.1 Globus 安全体系结构

用户代理完成，通过使一个进程基本上遵循第 2 个协议来令其关联的用户代理请求资源分配来实现。

Globus 安全体系结构中的第 4 个也是最后一个协议是用户使其自身在一个域内可知的方式。假设用户在一个域内有一个账号，那么需要确认的就是用户代理所持的系统范围的凭证自动转换为指定域能识别的凭证。该协议规定了用户进行全局凭证和局部凭证之间映射的注册方式，注册在位于该域的一个映射表中进行。

每个协议的特定细节都在(Foster 等 1998)中进行描述。这里我们强调的是，如上所述，Globus 安全结构反映了其安全策略。用于实现该结构的机制，特别是上面所提到的协议，对许多分布式系统都是共同的，这一点将在本章中详细讨论。设计安全的分布式系统的困难并不是主要由安全机制的问题引起的，而是由如何使用那些机制以实现安全策略的决策引起的。在下一节中，我们考虑这样的一些设计决策。

9.1.2 设计问题

一个分布式系统，或者说任何计算机系统，都必须提供安全服务，使得范围广阔的安全策略可以通过这些服务来实现。实现多方面的安全服务时需要考虑很多重要的设计问题。在下面的内容中，我们讨论下面 3 个问题：控制的焦点、安全机制分层和简洁性(Gollmann 2006)。

9.1.2.1 控制的焦点

考虑一个(可能是分布式的)应用程序的保护时，基本上可以遵循 3 种不同的方法，如

图9.2所示。第一种方法是把注意力直接集中于对与该应用程序相关联的数据的保护。我们使用“直接”这个词的意思是不考虑可能在数据项上执行的各种各样的操作，主要关心的是要确保数据完整性。通常，这种类型的保护出现在一些数据库系统中，在这些系统中可以规定各种完整性限制，每次修改一个数据项时系统自动检查这些限制（Doorn 和 Rivero 2002）。

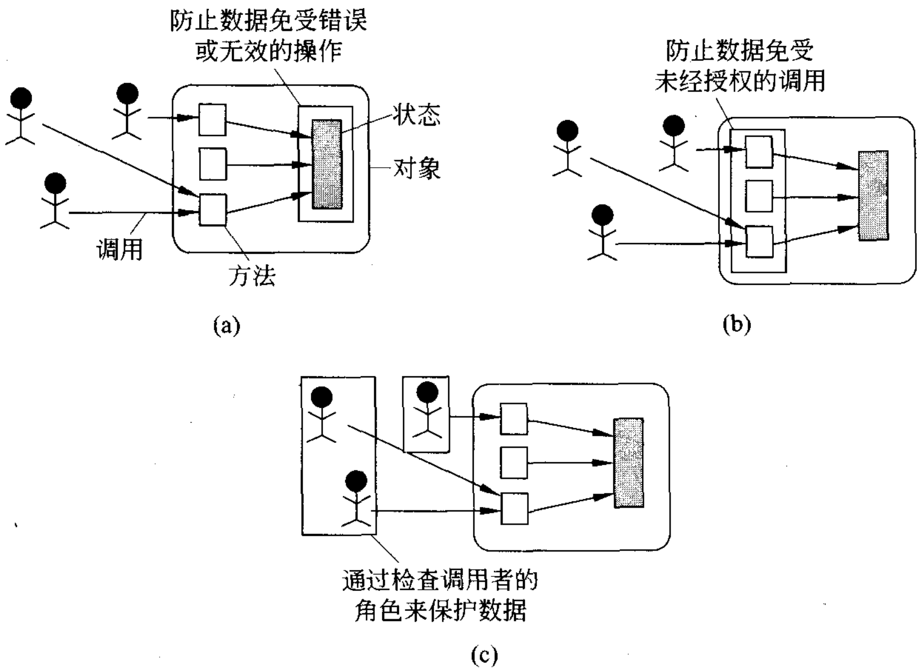


图 9.2 三种防止安全威胁的保护方法

(a) 防止无效操作的保护；(b) 防止未经授权调用的保护；(c) 防止未经授权用户的保护

第二种方法是把注意力集中于要访问特定数据或资源时，通过明确指定可以调用的操作，以及操作由谁完成来进行的保护上。在这种情况中，控制焦点与访问控制机制紧密相关，这一内容我们将在本章的后面部分进行详细讨论。例如，在一个基于对象的系统中，可以决定为客户可用的每个方法指定允许调用该方法的客户。换句话说，访问控制方法可以应用于一个对象所提供的完整接口或整个对象本身。因此这种方法考虑了访问控制的各种粒度。

第三种方法是把注意力直接集中于用户，通过采取一些措施使得只有指定的人可以访问该应用程序，而不考虑其要执行的操作来实现。例如，银行中的一个数据库可能通过拒绝除高级管理人员外任何人的访问来获得保护。另一个例子是，在许多大学中，限制特定的数据和应用程序只能由教职员工使用，而不允许学生访问。实际上，控制的焦点在于定义用户具有的**角色**（role）。一个用户的角色经过检验之后，就可以确定他对一个资源的访问是允许还是被拒绝。作为安全系统设计的一部分，定义人们可能具有的角色并提供支持基于角色的访问控制的机制是必要的。在本章的后面部分我们将回到角色这个话题。

9.1.2.2 安全机制的分层

安全系统设计的一个重要问题是要决定应将安全机制放置在哪一层。在这一环境中的层次与系统逻辑组织的层次相关。例如，计算机网络通常按照一些参考模型分层组织，正如

我们在第4章中所讨论的那样。在第1章中,我们介绍了由应用程序、中间件、操作系统服务和操作系统核心这些单独的层组成的分布式系统的结构。将计算机网络和分布式系统分层的组织结合起来,一般会导致如图9.3所示的结果。

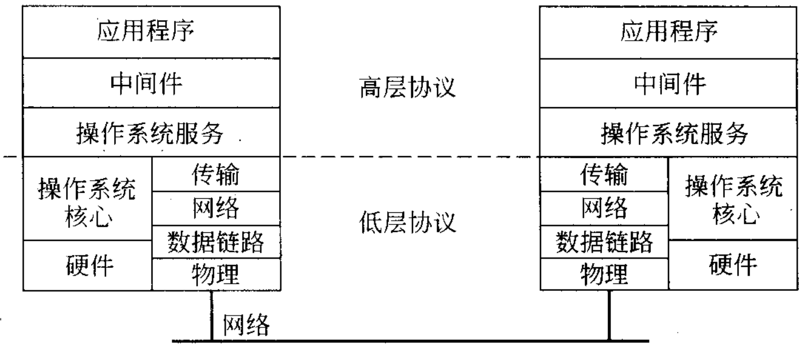


图9.3 分布式系统的多层逻辑结构

实际上,图9.3将通用服务与通信服务分离开来。这一分离对理解分布式系统中的分层安全,尤其是信任的概念非常重要。信任和安全之间的差别很重要。一个系统或者是安全的,或者是不安全的(考虑各种概率的措施),但一个客户是否认为一个系统是安全的,则是一个信任问题(Bishop 2003)。安全机制放置在哪一层取决于客户对特定层中服务的安全程度所具有的信任度。

作为一个例子,考虑位于不同站点的一个组织,这些站点通过诸如**交换式多兆位数据服务**(switched multimegabit data service,SMDS)的通信服务连接起来。一个SMDS网络可以看作一个链接层主干网,该网络连接位于地理上可能分散的地点的多个局域网,如图9.4所示。

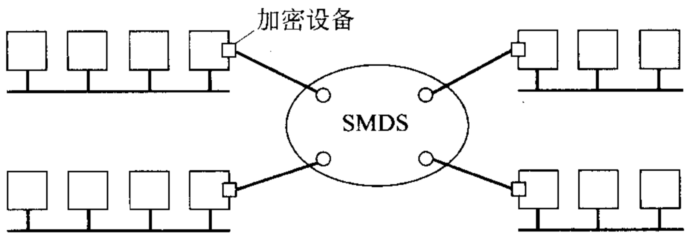


图9.4 通过广域主干服务连接的若干站点

安全可以通过在每个SMDS路由器处放置加密设备来提供,如图9.4所示。这些设备自动对站点之间发送的数据包进行加密和解密,但并不另外对同一站点的主机之间提供安全通信。如果站点A的Alice向站点B的Bob发送一条消息,并且担心她的消息被人窃听,那么她就必须至少信任站间通信的加密工作正常。例如,这意味着她必须信任两个站点的系统管理员都已采取适当措施来防止设备损坏。

现在假设Alice并不信任站间通信的安全。于是她可能决定采取自己的措施,即使用一个传输层的安全服务,例如SSL。SSL代表**安全套接字协议层**(secure socket layer),并且可以用于在一个TCP连接上安全地发送消息。我们将在第12章讨论Web中的安全时介绍SSL的细节问题。要注意的一个重要事项是SSL允许Alice建立与Bob的安全连接。所有传输级的消息都将被加密——SMDS级的也是一样,但这对Alice无关紧要。在这一

事例中，Alice 必须信任 SSL。换句话说，她相信 SSL 是安全的。

在分布式系统中，安全机制通常位于中间件层。如果 Alice 不信任 SSL，那么她可能希望使用一个本地的安全 RPC 服务。此外，她还必须信任这一 RPC 服务能够完成其许诺的任务，例如不泄露信息或适当地对客户和服务器进行身份认证。

位于分布式系统的中间件层的安全服务仅在其所依赖的安全服务确实安全的情况下才是可以信任的。例如，如果一个安全 RPC 服务的部分实现依靠 SSL，那么对该 RPC 服务的信任就取决于对 SSL 的信任程度。如果 SSL 的安全是不可靠的，就不可能信任该 RPC 服务的安全性。

9.1.2.3 安全机制的分布性

关于信任的服务之间的相关性导致了**信任计算库**(trusted computing base，TCB)概念的出现。TCB 是一个(分布式)系统中所有安全机制的集合，这些机制用于执行一个安全策略。TCB 越小越好。如果在现有的网络操作系统上将一个分布式系统作为中间件建立，那么其安全性就取决于作为基础的本地操作系统的安全性。换句话说，分布式系统中的 TCB 可以包括不同主机上的本地操作系统。

我们来考虑分布式文件系统中的文件服务器的情况。这样的一个服务器可能需要依赖其本地操作系统所提供的各种保护机制。这些机制不仅包括保护文件免受文件服务器外的进程访问的机制，还包括保护该文件服务器免受恶意破坏的机制。

因此基于中间件的分布式系统需要有对其依赖的现有本地操作系统的信任。如果不存在这样的信任，则本地操作系统的部分功能性就可能需要合并到分布式系统本身中。考虑一个微核操作系统，其中的许多操作系统服务作为普通用户进程运行。例如，在这种情况下，文件系统可以由一个为了适合分布式系统特定需求而进行裁减的系统(包括各种安全措施)完全取代。

与这种方法一致的是根据所需要的安全性将服务分布在不同的机器上，以便将安全服务与其他类型的服务分开。例如，对一个安全的分布式文件系统来说，有可能通过把服务器放在一个具有可信操作系统并可能运行专用安全文件系统的机器上，从而将该文件服务器与客户隔离开，让客户及其应用程序位于不可信的机器上。

这一分离有效地将 TCB 简化为一个相对较少数量的机器和软件组件。通过随后保护那些机器免受外界的安全攻击，能够提升对分布式系统安全的全面信任。正如(Neumann 1995)中所描述的，**安全系统组件简化接口**(reduced interfaces for secure system components，RISSC)方法遵循了防止客户及其应用程序直接访问关键服务的原则。在 RISSC 方法中，任何安全性非常关键(security-critical)的服务器都放置在一台单独的机器上，与使用低级安全网络接口的终端用户系统隔离开，如图 9.5 所示。客户及其应用程序在不同机器上运行，并且仅能通过这些网络接口访问受保护的服务器。

9.1.2.4 简单性

另一个与决定将安全机制放在哪一层有关的重要设计问题是简单性问题。一般认为设计一个安全的计算机系统是一项困难的任务。因此，如果一个系统设计者可以使用一些易于理解且工作可靠的简单机制，设计工作就会较容易实现。

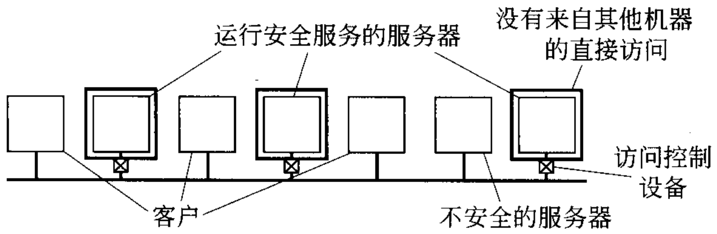


图 9.5 应用于安全分布式系统的 RISSC 原理

不幸的是，对实现安全策略来说简单机制通常不够充分。我们来再次考虑一下前面讨论的 Alice 希望向 Bob 发送一条消息的情况。链接层加密是一个简单且易于理解的机制，它用来防止对站点间消息通信的窃听。然而，如果 Alice 希望确保只有 Bob 能够接收到她的消息，就需要更多机制。在这种情况下，就需要用户级身份认证服务，并且 Alice 可能需要了解此类服务的工作方式以便投入其信任。因此尽管许多安全服务都是高度自动化且对用户隐藏的，但用户级身份认证可能至少还需要对加密密钥的概念的理解以及对诸如证书这样的机制的认识。

在其他情况下，应用程序自身本质上是复杂的，引入安全只会使事情变得更糟。复杂安全协议的一个实例应用领域（如我们在本章后面要讨论的）是一个数字支付系统。一次数字支付必须由多方进行通信才能完成，这就导致了数字支付的复杂性。在这些情况下，用于实现协议的基本机制要相对简单且易于理解，这一点很重要。简单性有助于最终用户对应用程序投入信任，更重要的是，对使设计者确信系统没有安全漏洞有很大贡献。

9.1.3 加密

在分布式系统中，安全的基本措施是使用加密技术。应用这些技术的基本思想是很简单的。我们来考虑一个发送方 S 希望向接收方 R 传送消息 m 的情况。要保护该消息免受安全威胁，发送方首先将其**加密**(encrypt)为一个难以理解的消息 m'，然后将 m' 发送到 R。反过来，R 必须将接收到的消息**解密**(decrypt)为其原始格式 m。

加密和解密通过使用以密钥为参数的加密算法实现，如图 9.6 所示。发送的消息的原始格式称为**明文**(plaintext)，如图 9.6 中的 P 所示；已加密的格式称为**密文**(ciphertext)，由 C 表示。

要描述为分布式系统建立安全服务所使用的各种安全协议，一个将明文、密文和密钥联系起来的符号表示法是很有用的。遵循普通的符号表示惯例，我们将使用 $C=E_K(P)$ 来表示通过使用密钥 K 对明文 P 进行加密获得密文 C。同样，使用 $P=D_K(C)$ 来表示对使用密钥 K 的密文 C 的解密，其结果为明文 P。

回到在图 9.6 中显示的例子。以密文 C 传输消息时，存在三种需要我们防止的不同攻击，加密对防止这些攻击是有帮助的。首先，**入侵者**(intruder)可能在发送方或接收方都不知道发生窃听的情况下窃听该消息，当然，如果传输的消息已经以没有正确密钥就不易解密的方式进行了加密，那么窃听就是无用的，入侵者将只能看到不能理解的数据（顺便说一下，仅一个消息正在传输这一事实有时就足以让入侵者得出结论。例如，如果在一次世界危机期间到白宫的通信流量突然下降到零，而到 Colorado 的一个特定山脉的通信流量以相同数

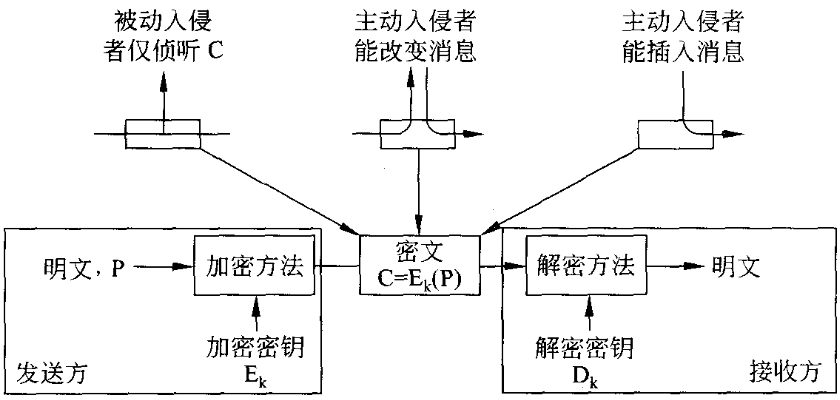


图 9.6　通信中的入侵者和窃听者

量增加，那么，了解到这一点就可能获得有用的信息）。

需要处理的第二种类型的攻击是修改消息。修改明文是很容易的，修改经过适当加密的密文则困难得多。因为在能够有针对性地修改消息前，入侵者首先必须对该消息进行解密。此外，他还必须重新适当地对该消息进行加密，否则接收方就可能注意到消息已经被篡改。

第三种类型的攻击是当入侵者向通信系统中插入加密过的消息时，他们企图令 R 相信这些消息来自 S。此外，正如我们将在本章后面所看到的，加密可以帮助防止这样的攻击。注意，如果入侵者能够修改消息，那么他同样能够插入消息。

不同加密系统之间存在一个基本区别，这一区别基于加密与解密密钥是否相同。在一个**对称加密系统**（symmetric cryptosystem）中，使用相同的密钥进行一个消息的加密和解密，换句话说：

$$P = D_K(E_K(P))$$

对称加密系统还称为密钥或共享密钥系统，因为需要发送方和接收方共享相同的密钥，要确保这种保护正常工作，这一共享密钥必须保密，不允许其他人看到此密钥。我们将使用符号 $K_{A,B}$来表示 A 和 B 共享的一个密钥。

在一个非对称加密系统中，加密和解密所使用的密钥是不同的，但两个密钥一起构成唯一的一对，换句话说，存在一个单独的密钥 K_E用于加密和一个用于解密的密钥 K_D，例如

$$P = D_{K_D}(E_{K_E}(P))$$

非对称加密系统中的一个密钥是保密的，另一个是公开的。出于这种原因，非对称加密系统还称为公钥系统。在后面的内容中，我们使用符号 K_A^+ 来表示属于 A 的一个公钥，K_A^- 作为其相应的私钥。

我们来预先考虑一下本章后面有关安全协议的详细讨论。加密与解密密钥中的哪一个会实际作为公钥取决于两个密钥的使用方式。例如，如果 Alice 希望向 Bob 发送一个机密消息，那么她应该使用 Bob 的公钥加密该消息。因为 Bob 是唯一持有解密私钥的人，他同样也是唯一能够解密该消息的人。

另一方面，假设 Bob 希望确切知道他刚刚收到的消息确实来自 Alice。在这种情况下，

Alice 可以保持其加密密钥为个人专用，并用它来加密其发送的消息。如果 Bob 使用 Alice 的公钥能够成功解密消息（并且该消息中的明文具有足够的信息可使其对 Bob 有意义），那么他就知道该消息确实来自 Alice，因为解密密钥与加密密钥是唯一绑定的。下面我们将回来详细介绍这样的算法。

在分布式系统中，密码学的一个最终应用是**散列函数**（hash function）的使用。一个散列函数 H 将任意长度的消息 m 作为输入，并产生一个具有固定长度的位串 h 作为输出：

$$h=H(m)$$

一个散列 h 相当于在通信系统中向一条消息添加的附加位，这些附加位用于错误检测，例如**循环冗余检查**（cyclic-redundancy check，CRC）。

加密系统中使用的散列函数具有很多属性。首先，这些函数是**单向函数**（one-way function），意思是不可能计算出与已知输出 h 相应的输入 m。另一方面，从 m 计算 h 是很容易的。其次，这些函数具有**弱抗冲突性**（weak collision resistance），意思是给定一个输入 m，其相关输出 $h=H(m)$，不可能通过计算找到另一个不同的输入 $m'\neq m$，使 $H(m)=H(m')$。最后，加密散列函数还具有**强抗冲突性**（strong collision resistance），这意味着仅给定 H 时，不可能计算出任何两个不同的输入值 m 和 m′，使 $H(m)=H(m')$。

相似的性质适用于任何加密函数 E 及所使用的密钥上。也就是说，对任何加密函数 E 来说，给定明文 P 和相关的密文 $C=E_K(P)$，都不可能通过计算找到密钥 K。同样，类似于抗冲突性，给定明文 P 和密钥 K 时，应该不可能找到另一个密钥 K′，使 $E_K(P)=E_{K'}(P)$。

为加密系统设计算法的技术和科学有很长的历史（Kahn 1967），建立安全系统通常异乎寻常地困难，或者甚至不可能（Schneier 2000）。详细讨论这些算法中的任何一种都超出了本书的范围。然而，为了对计算机系统中的加密提供一些概念，我们现在简要介绍三个有代表性的算法。有关这些算法和其他加密算法的详细信息可在（Schneier 2003，Menezes 等 1996，Schneier 1996）中找到。

在我们详细研究各种协议的细节之前，图 9.7 总结了我们使用的符号和缩写。

符号	描述
$K_{A,B}$	A 和 B 共享的密钥
K^+_A	A 的公钥
K^-_A	A 的私钥

图 9.7 本章中使用的符号

9.1.3.1 对称加密系统：DES

我们的第一个加密算法的例子是用于对称加密系统的**数据加密标准**（data encryption standard，DES）。DES 设计成运行在 64 位数据块上。数据块在 16 轮内转换为一个加密的（64 位）输出块，每一轮使用一个不同的 48 位密钥进行加密。这 16 个密钥中的每一个都来源于一个 56 位的主密钥，如图 9.8(a)所示。一个输入块在开始其 16 轮的加密之前，首先要进行初始排列，其逆随后应用于加密过的输出上，导致最终的输出数据块。

每个加密轮 i 将前一轮 i−1 产生的 64 位数据块作为其输入，如图 9.8(b)所示。这 64 位分为左边部分 L_{i-1} 和右边部分 R_{i-1}，每部分包含 32 位。右边部分作为下一轮的左边部分，也就是说，$L_i=R_{i-1}$。

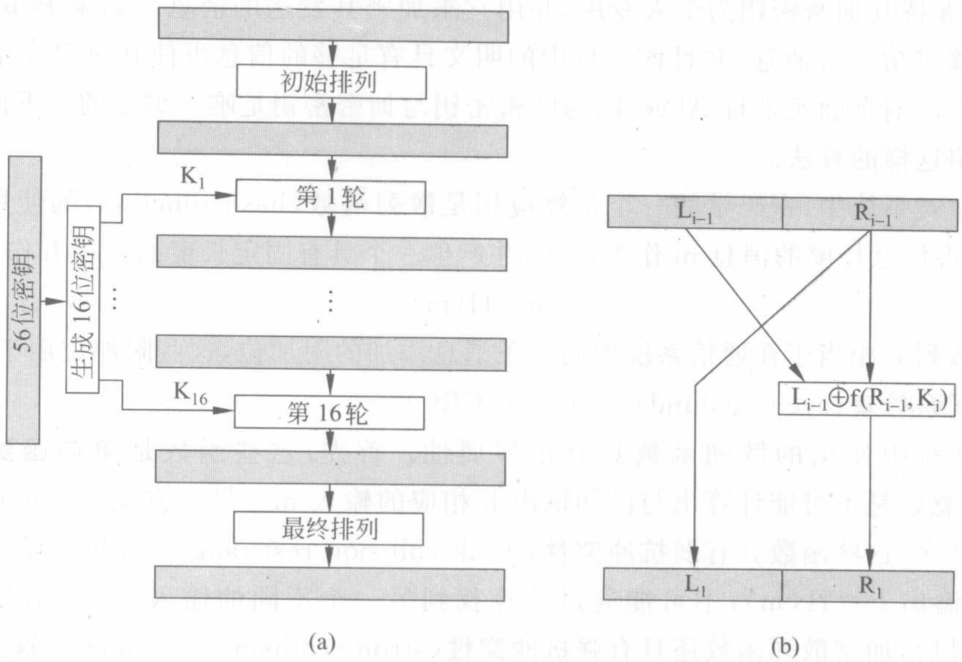


图　9.8

(a) DES 的原理；(b) 一个加密轮次的概要

困难的工作在函数 f 中完成。此函数将一个 32 位的数据块 R_{i-1} 作为输入，加上一个 48 位的密钥 K_i，产生一个 32 位的数据块与 L_{i-1} 进行 XOR 运算以产生 R_i。（XOR 是异或运算的缩写。）函数 f 首先将 R_{i-1} 扩展为 48 位的数据块并与 K_i 进行 XOR 运算。结果被划分为 8 块，每块 6 位。然后将每块输入到不同的 S 盒中，S 盒是一种运算，将有 64 种可能的 6 位输入的每一种都替换为有 16 种可能的 4 位输出的一种。然后每块 4 位的 8 个输出块结合为一个 32 位值并重新排列。

第 i 轮的 48 位密钥 K_i 是从 56 位的主密钥中得到的，其过程如下所述。首先，排列主密钥并将其分为两个 28 位的等分。对每轮来说，每一半首先都左移一位或两位，之后就取出 24 位。先取出的 24 位与从移动后的另一半中取出的 24 位在一起，就构成了一个 48 位的密钥。一个加密轮次的细节如图 9.9 所示。

DES 的原理是非常简单的，但使用分析方法难以破解该算法。正如最近所证明的，通过简单地搜索能够使用的密钥进行的粗野攻击已经变得很容易防范了。在一个具有不同密钥的特殊的加密-解密-加密模式中三次使用 DES（又称为**三重 DES**(triple DES)）更安全，且仍在使用中(Barker 2004)。

使 DES 难以通过分析进行攻击的原因是设计背后的基本原理从来没有清晰地解释过。例如，已经知道，采用其他 S 盒而不用当前标准中所使用的那些，会使该算法很大程度上易于破解（请参阅(Pfleeger 2003)以获得 DES 的一个简要分析）。设计的基本原理和 S 盒的使用仅在 20 世纪 90 年代设计出“新的”攻击模型后才公布。DES 证实对这些攻击非常有抵抗力，且其设计者透露 1974 年他们开发 DES 时就已经知道新发明的攻击模型了(Coppersmith 1994)。

多年以来，DES 已经作为标准加密技术使用了，但目前正处在被 128 位的 Rijndael 算法数据块取代的过程中。还有具有更大密钥和更大数据块的变体。该算法设计得非常快，以至于可以在智能卡上实现，这就形成了一个日益重要的密码学应用领域。

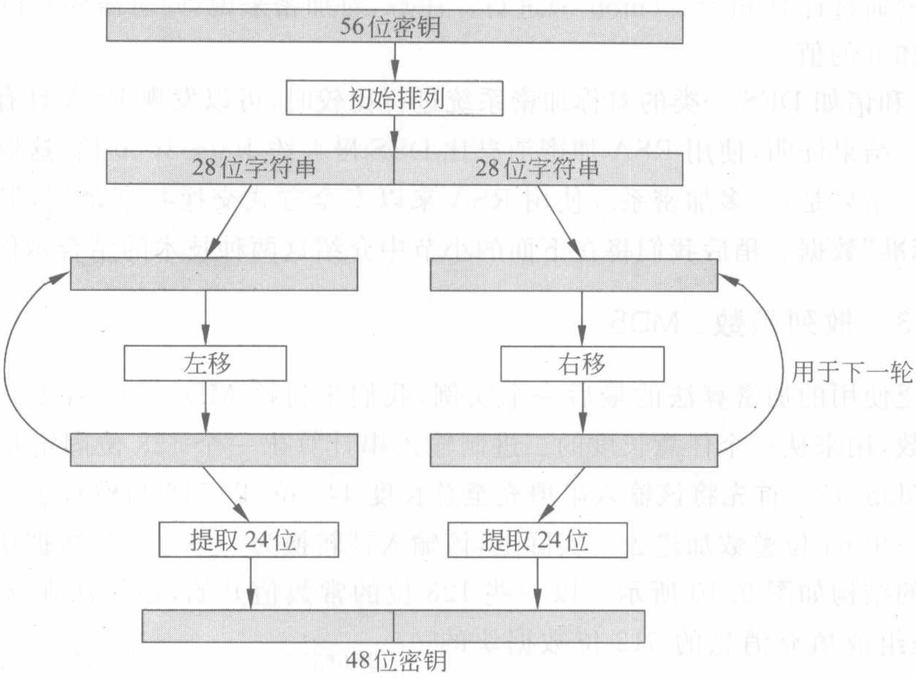


图 9.9 DES 中每轮密钥生成的细节

9.1.3.2 公钥加密系统：RSA

我们的第二个加密算法实例广泛应用于公钥系统，这就是 **RSA**。RSA 按其发明者 Rivest、Shamir 和 Adieman 名字的首字母命名(1978)。RSA 的安全性来自这样一个事实，即没有任何已知的方法能够有效地找到大数的**素因子**(prime factor)。可以看出每个整数都可以写成素数的乘积。例如，2100 可以写成：

$$2100=2\times2\times3\times5\times5\times7$$

这样就使 2、3、5 和 7 成为 2100 的素因子。在 RSA 中，私钥和公钥都是由非常大的素数(由上百个十进制数字组成)构成的。结果证明，破解 RSA 相当于寻找这两个素数。尽管数学家已经致力于这个问题很长时间，但到目前为止显示的结果不可能计算出来。

产生私钥和公钥需要 4 个步骤：

(1) 选择两个非常大的素数，p 和 q。

(2) 计算 $n=p\times q$ 以及 $z=(p-1)\times(q-1)$。

(3) 选择一个与 z 有关的素数 d。

(4) 计算数 e 使得 $e\times d=1 \bmod z$。

其中一个数 d 可以随后用于解密，而 e 则用于加密。这两个数中只能公开一个，取决于使用该算法的目的。

我们来考虑 Alice 希望将其发送给 Bob 的消息保密的情况。换句话说，她希望确保除 Bob 外没有人能够窃听和读取其发送给 Bob 的消息。RSA 认为每个消息 m 只是一个比特串。首先将每个消息分为固定长度的数据块，其中每个块 m_i 被解释为一个二进制数，应该位于区间 $0\leqslant m_i<n$ 内。

要加密消息 m，发送方为每个块 m_i 计算值 $c_i=m_i^e \pmod n$，然后将该值发送给接收方。

接收方的解密通过计算 $m_i = c_i^d \pmod n$ 进行。注意，对加密来说，同时需要 e 和 n，而解密则需要知道 d 和 n 的值。

将 RSA 和诸如 DES 一类的对称加密系统进行比较时，可以发现 RSA 具有计算上更为复杂的缺点。结果证明，使用 RSA 加密消息比 DES 慢大约 100～1000 倍，这取决于所使用的实现技术。结果是，许多加密系统使用 RSA 来以安全方式交换共享密钥，但很少用它来实际加密“标准”数据。稍后我们将在下面的小节中介绍这两种技术的结合示例。

9.1.3.3 散列函数：MD5

作为广泛使用的加密算法的最后一个实例，我们来讨论 MD5（Rivest 1992）。**MD5** 是一个散列函数，用来从一个任意长度的二进制输入串计算出一个 128 位固定长度的**消息摘要**（message digest）。首先将该输入串填充至总长度 448 位（以 512 为模），然后原始比特串的长度作为一个 64 位整数加进去。实际上，该输入被转换为一串 512 位数据块。

该算法的结构如图 9.10 所示。以一些 128 位的常数值开始，该算法在 k 个阶段中进行，其中 k 是组成填充消息的 512 位数据块的数量。在每个阶段中，用来自填充消息的 512 位数据块和前一阶段计算出的 128 位摘要计算出一个 128 位的摘要。MD5 中的一个阶段由 4 轮计算组成，其中每轮使用下面的 4 个函数中的一种：

$F(x,y,z) = (x \text{ AND } y) \text{ OR } ((\text{NOT } x) \text{ AND } z)$

$G(x,y,z) = (x \text{ AND } z) \text{ OR } (y \text{ AND } (\text{NOT } z))$

$H(x,y,z) = x \text{ XOR } y \text{ XOR } z$

$I(x,y,z) = y \text{ XOR } (x \text{ OR } (\text{NOT } z))$

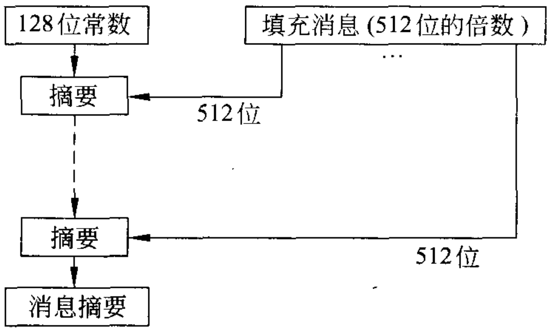


图 9.10 MD5 的结构

这些函数的每一个都对 32 位变量 x、y 和 z 进行运算。为了说明这些函数的使用方式，请考虑来自填充消息的一个 512 位数据块 b，该数据块正在第 k 阶段进行处理。数据块 b 被划分为 16 个 32 位的子块 $b_0, b_1, \cdots, b_{15}$。在第一轮中，16 次迭代使用函数 F 来改变 4 个变量（分别表示为 p、q、r 和 s），如图 9.11 所示。这些变量传到各自的下一轮，一个阶段结束后，继续传到下一个阶段。总共有 64 个预定义常量 C_i。符号 x<<<n 表示左移，x 中的位向左移动 n 个位置，左边移出的位放在最右边的位置上。

第 1～8 次迭代	第 9～16 次迭代
$p \leftarrow (p+F(q, r, s)+b_0+C_1)<<<7$	$p \leftarrow (p+F(q, r, s)+b_8+C_9)<<<7$
$s \leftarrow (s+F(p, q, r)+b_1+C_2)<<<12$	$s \leftarrow (s+F(p, q, r)+b_9+C_{10})<<<12$
$r \leftarrow (r+F(s, p, q)+b_2+C_3)<<<17$	$r \leftarrow (r+F(s, p, q)+b_{10}+C_{11})<<<17$
$q \leftarrow (q+F(r, s, p)+b_3+C_4)<<<22$	$q \leftarrow (q+F(r, s, p)+b_{11}+C_{12})<<<22$
$p \leftarrow (p+F(q, r, s)+b_4+C_5)<<<7$	$p \leftarrow (p+F(q, r, s)+b_{12}+C_{13})<<<7$
$s \leftarrow (s+F(q, q, r)+b_5+C_6)<<<12$	$s \leftarrow (s+F(q, q, r)+b_{13}+C_{14})<<<12$
$r \leftarrow (r+F(s, p, q)+b_6+C_7)<<<17$	$r \leftarrow (r+F(s, p, q)+b_{14}+C_{15})<<<17$
$q \leftarrow (q+F(r, s, p)+b_7+C_8)<<<22$	$q \leftarrow (q+F(r, s, p)+b_{15}+C_{16})<<<22$

图 9.11 MD5 中一个阶段的第一轮期间的 16 次迭代

第二轮以相似的方式使用函数G，而H和I则分别在第三轮和第四轮中使用。因此每一步都由64次迭代组成，随后开始下一个阶段，而且p、q、r和s在该点所具有的值也带入下一阶段。

9.2 安全通道

在前面的章节中，我们通常使用客户-服务器模型作为一种便利的方式来组织分布式系统。在此模型中，服务器可能是分布式和复制的，同时也可能充当与其他服务器相关的客户。考虑分布式系统中的安全性时，重新以客户和服务器的观点来思考是很有用的。特别是，使一个分布式系统安全性从本质上归结为两个主要问题。第一个问题是如何使客户与服务器之间的通信保持安全。安全通信需要对通信各方进行身份认证，同时还要确保消息完整性和可能的机密性。作为这一问题的一部分，我们还需要考虑保护一组服务器内的通信。

第二个问题是授权问题，即服务器一旦接受来自客户的请求，它将如何查明该客户是否得到授权使请求得以执行？身份认证与资源访问控制问题相关，这一问题我们将在下一小节中详细讨论。在本节中，我们集中于分布式系统内通信的保护。

客户和服务器之间通信的保护问题可以认为是在通信各方之间建立一个**安全通道**(secure channel)的问题(Voydock和Kent 1983)。安全通道保护发送方和接收方免受对消息的窃听、修改和伪造的攻击。对防止中断的保护不是必要的。保护消息免受窃听是通过确保机密性实现的，安全通道可确保入侵者不能窃听到其消息。防止入侵者的修改和伪造是通过相互身份认证和消息完整性的协议实现的。在下面的内容中，我们首先讨论可以用于身份认证的各种协议，这些协议使用对称加密系统和公钥加密系统。身份认证的基础逻辑的详细描述可以在(Lampson等1992)中找到。我们将分别讨论机密性和消息完整性。

9.2.1 身份认证

在详细阐述各种身份认证的细节之前，值得注意的是身份认证和消息完整性相互之间不能脱离开。例如，考虑一个分布式系统，该系统支持两个通信方的身份认证，但不提供确保消息完整性的机制。在这样的一个系统中，Bob可以确切知道Alice是否为m的发送者。然而，如果Bob不能获得m在传输期间没有被修改的保证，那么即使他知道Alice发送了(原始版本的)m又有什么用呢？

同样，假设系统只支持消息完整性，但不存在身份认证机制。Bob接收到一个消息宣称他刚刚在彩票中赢得了1 000 000美元时，如果他不能检验该消息是否是由该彩票的组织者发送的，那么他能高兴到什么程度呢？

因此，身份认证和消息完整性应该结合在一起。在许多协议中，这一结合一般按下面的方式进行。我们再一次假设Alice和Bob希望进行通信，并且Alice主动建立通道。Alice通过向Bob发送一条消息，或者另外向将帮助建立通道的一个可靠的第三方发送消息来开始通信。通道建立后，Alice就确切知道她在与Bob通话，并且Bob也确切知道他在与

Alice 通话，于是他们可以交换消息了。

要确保进行身份认证之后交换的数据消息的完整性，常见的方法是依靠会话密钥使用密钥加密。**会话密钥**(session key)是一个共享密钥，用于为完整性和可信的机密性而对消息进行加密。这样的密钥一般仅在通道存在的时候使用。通道关闭时，就丢弃（或者实际上安全地破坏）与其相关的会话密钥。下面我们回来讨论会话密钥。

9.2.1.1 基于共享密钥的身份认证

我们从考虑一个基于已经在 Alice 和 Bob 之间共享的安全密钥的身份认证协议开始，稍后在本章中讨论双方试图以安全方式获得一个共享密钥的实际方法。在该协议的描述中，Alice 和 Bob 分别缩写为 A 和 B，且其共享密钥表示为 $K_{A,B}$。该协议采用了一种普通的方法，即一方向另一方质询一个响应，而该响应只有在另一方知道共享密钥的情况下是正确的。这种解决方法也称为**质询-响应协议**(challenge-response protocol)。

在基于共享密钥的身份认证的情况中，该协议的进行如图 9.12 所示。首先，Alice 将其身份（消息 1）发送给 Bob，指出她希望在两人之间建立一个通信通道，随后 Bob 向 Alice 发送一个质询 R_B（消息 2）。这样的一个质询可以采取随机数的形式。需要 Alice 使用她与 Bob 共享的密钥 KA',B 加密该质询，并将加密过的质询返回给 Bob。此响应在图 9.12 中被表示为消息 3，它包含 $K_{A,B}(R_B)$。

Bob 接收到对其质询 R_B 的响应 $K_{A,B}(R_B)$ 时，可以再次使用该共享密钥对该消息进行解密以查看其中是否包含 R_B。如果包含，那么他就知道 Alice 在另一边，其他谁能够首先用 $K_{A,B}$ 加密 R_B？换句话说，Bob 现在已经检验出他确实在和 Alice 交谈。然而，注意，Alice 还没有检验出通道的另一边确实是 Bob。因此，她发送一个质询 R_A（消息 4），Bob 通过返回 $K_{A,B}(R_A)$ 响应该消息（消息 5）。Alice 使用 $K_{A,B}$ 解密该消息并看到 R_A 时，她就知道在与 Bob 交谈。

安全中一个较困难的问题是设计能实际工作的协议。为了描述事情发生错误是多么容易，我们来考虑一个“最优化”的身份认证协议，其中的消息数量从 5 减至 3，如图 9.13 所示。基本思想是如果 Alice 最终希望用任何方式质询 Bob，那么她在建立通道时可能还会与其身份一起发送一个质询。同样，Bob 在单个消息中返回其对该质询的响应，以及自己的质询。

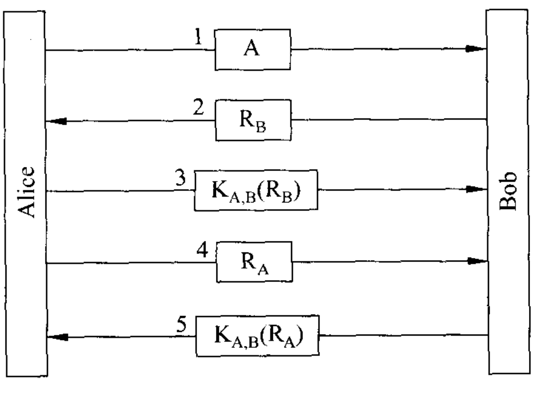


图 9.12 基于共享密钥的身份认证

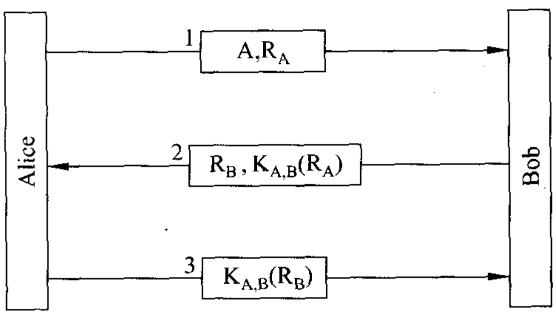


图 9.13 基于共享密钥，但使用 3 个而不是 5 个消息的身份认证

不幸的是，此协议不再工作。它可以轻易地被称为**反射攻击**(reflection attack)的行为打败。要解释这种攻击的工作方式，假设有一个名为 Chuck 的入侵者，在我们的协议中将其表示为 C。Chuck 的目标是与 Bob 建立一个通道，以使 Bob 相信他在与 Alice 交谈。如果 Chuck 正确响应由 Bob 发送的一个质询，例如，通过返回 Bob 发送的数字的加密版本，那么他就可以建立这个通道。在不知道 $K_{A,B}$ 的情况下，只有 Bob 可以进行这样的加密，而这恰恰正是 Chuck 哄骗 Bob 所做的事。

该攻击在图 9.14 中进行了解释。Chuck 通过发送一条包含 Alice 的身份标识 A 的消息以及一个质询 R_C 开始通信，Bob 用单条消息返回其质询 R_B 和响应 $K_{A,B}(R_C)$。这时，Chuck 需要通过返回给 Bob 的 $K_{A,B}(R_B)$ 来证明他知道该密钥。不幸的是，他没有 $K_{A,B}$。作为替代，他所做的就是试图建立第二条通道以使 Bob 为其进行加密。

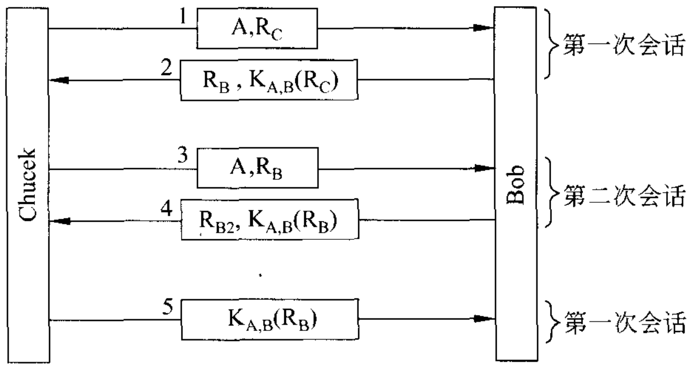


图 9.14 反射攻击

因此，Chuck 像以前 Alice 所做的那样用单条消息发送 A 和 R_B，但现在假定需要第二条通道。这在图 9.14 的消息 3 中显示。Bob 并没有认识到他自己先前已经使用过 R_B 作为质询，就用 $K_{A,B}(R_B)$ 和另一个质询 R_{B2} 进行响应(消息 4)。那时，Chuck 就具有了 $K_{A,B}(R_B)$，并通过返回包含 $K_{A,B}(R_B)$ 响应的消息 5 完成了第一个会话的建立，该响应最初是由消息 2 中发送的质询请求的。

如(Kaufman 等 2003)中所解释的，原始协议改编过程中所犯的错误之一是，协议新版本中的双方在协议的两个不同方向都使用相同的质询。一个较好的设计是永远为发起者和响应者使用不同的质询。例如，如果 Alice 总使用奇数且 Bob 总使用偶数，那么 Bob 接收到图 9.14 中的消息 3 中的 R_B 时，就会认识到发生了一些可疑的事情(不幸的是，此解决方法常遭受其他攻击，特别是称为"**中间人攻击**(man-in-the-middle-attack)"的攻击，该攻击在(Ferguson 和 Schneier 2003)中进行了介绍)。一般说来，让建立安全通道中涉及的双方同等地做很多事情并不是个好主意。

改编协议中违背的另一个原则是，Bob 以响应 $K_{A,B}(R_C)$ 的形式泄露了有价值的信息，而没有确切了解他的信息发送给何人。这一原则在原始协议中没有违背，其中 Alice 首先需要证实其身份，然后 Bob 希望传送她的加密信息。

还有一些其他原则，这些年被加密协议开发人员逐渐了解，我们将在下面讨论其他协议的时候介绍其中的一些。一个重要的教训是，将安全协议设计成能完成要其完成的任务通常比看起来困难很多。同样，正如我们上面所论述的那样，修改一个现有的协议以提高其性能影响其正确性。有关协议的设计原则的更多信息可以在(Abadi 和 Needham 1996)中

找到。

9.2.1.2 使用密钥分发中心的身份认证

使用共享密钥进行身份认证的一个问题是可扩展性。如果一个分布式系统包含N台主机，并且需要每台主机与其他(N－1)台主机共享一个密钥，那么该系统作为一个整体就需要管理N(N－1)/2个密钥，并且每台主机必须管理(N－1)个密钥。对较大的N来说，这将导致出现问题。一个替代办法是依赖一个**密钥分发中心**(key distribution center，KDC)使用集中式方法。这一KDC与每台主机共享一个密钥，但不需要主机对拥有共享密钥。换句话说，使用KDC需要我们管理N个密钥而不是N(N－1)/2个密钥，这无疑是一个改进。

如果Alice希望与Bob建立一条安全通道，那么她可以通过一个(可靠的)KDC的帮助来实现。整体思想是KDC向Alice和Bob两人分发一个密钥，他们可以使用该密钥进行通信，如图9.15所示。

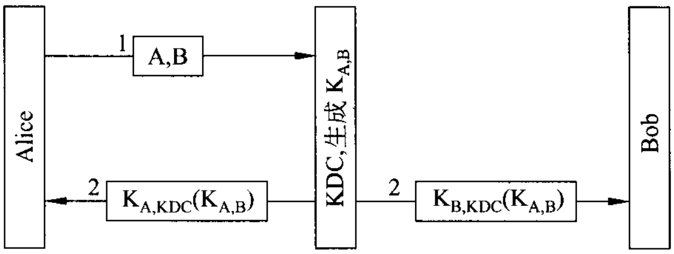


图9.15 使用KDC的原理

Alice首先向KDC发送一条消息，通知KDC她希望与Bob交谈。KDC返回一个包含她能够使用的共享密钥$K_{A,B}$的消息。该消息用Alice与KDC共享的密钥$K_{A,KDC}$进行了加密。此外，KDC也向Bob发送$K_{A,B}$，但此时使用与Bob共享的密钥$K_{B,KDC}$进行加密。

这种方法的主要缺点是Alice可能希望在Bob接收到来自KDC的共享密钥之前就开始与Bob建立一个安全通道。此外，需要KDC向Bob传送密钥来使Bob进入循环。如果KDC恰好将$K_{B,KDC}(K_{A,B})$传回给Alice，并令其关心与Bob的连接，就可以避开这些问题。这就导致了如图9.16中所示的协议。消息$K_{B,KDC}(K_{A,B})$也称为**票据**(ticket)。将此票据传给Bob是Alice的工作。注意Bob仍是唯一的一个能够有意识地使用该票据的人，因为他是除KDC外唯一知道该票据中所包含消息的解密方法的人。

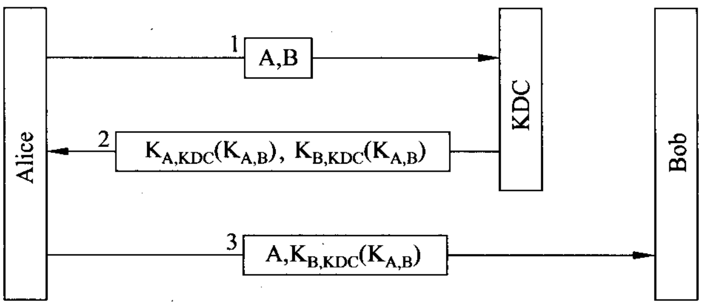


图9.16 使用票据并让Alice建立一个与Bob的通道

图 9.16 中所示的协议是使用 KDC 的一个身份认证协议的变体，该协议名为 Needham-Schroeder 身份认证协议，按其发明者(Needham 和 Schroeder 1978)名字命名。该协议的一个不同变体应用于我们稍后要描述的 Kerberos 系统中。Needham-Schroeder 协议，如图 9.17 所示，是一个多向质询-响应协议，其工作方式如下所述。

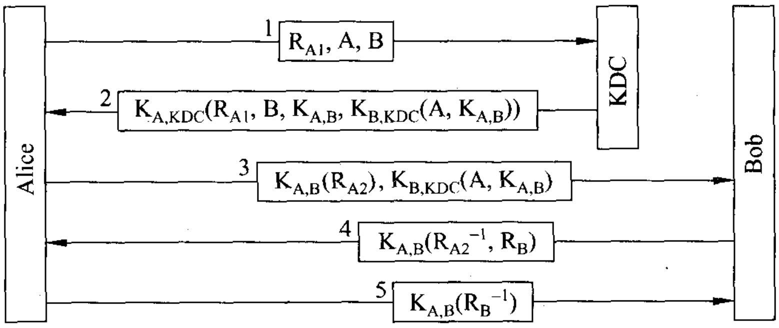


图 9.17 Needham-Schroeder 身份认证协议

当 Alice 希望与 Bob 建立一个安全通道时，她就向 KDC 发送一个包含质询 RA，以及其身份标识 A，当然还有 Bob 的身份标识。KDC 通过给予其票据 $K_{B,KDC}(K_{A,B})$以及她可以随后与 Bob 共享的密钥 $K_{A,B}$进行响应。

Alice 发送给 KDC 的质询 R_{A1} 连同其与 Bob 建立通道的请求还被称为“现时”。**现时**(nonce)是一个仅使用一次的随机数，比如从一个非常大的随机数集合中选取一个。现时的主要用途是将两条消息唯一地互相联系起来，在本例中是消息 1 和消息 2。特别是，通过在消息 2 中再次加入 R_{A1}，Alice 将确切知道消息 2 是作为消息 1 的响应发送的，而不是较早的消息的重放。

为了理解即将发生的问题，假设我们不使用现时，并且 Chuck 已经窃取了 Bob 的一个原先使用的密钥，即 $k^{old}_{B,KDC}$。此外，Chuck 已经窃听到了一个旧的响应 $K_{A,KDC}(B, K_{A,B}, K^{old}_{B,KDC}(A, K_{A,B}))$，该响应是 KDC 对来自 Alice 的与 Bob 交谈的早先请求返回的响应。这时，Bob 希望已经与 KDC 协商了一个新的共享密钥。然而，Chuck 耐心地等待，直到 Alice 再一次请求建立与 Bob 的安全通道。那时，他就重放旧的响应，哄骗 Alice 相信她正与 Bob 交谈，因为这样他就可以解密该票据并证实其知道共享密钥 $K_{A,B}$。

通过加入一个现时，这样的攻击就不可能成功，因为重放较早的消息将立即被发现。特别是，响应消息中的现时不会与原始请求中的现时相匹配。

消息 2 也包含 Bob 的身份标识 B。通过加入 B，KDC 保护 Alice 免受下列攻击。假设消息 2 中遗漏了 B。在这种情况下，Chuck 可以通过用其自身的身份 C 取代 Bob 的身份来修改消息 1。然后 KDC 会以为 Alice 希望建立到 Chuck 的安全通道，并因此进行响应。一旦 Alice 希望与 Bob 联系，Chuck 就窃听该消息，并哄骗 Alice 相信她正与 Bob 交谈。通过从消息 1 到消息 2 复制另一方的身份，Alice 就会立即检测到她的请求已经被修改。

KDC 将该票据传给 Alice 后，Alice 和 Bob 之间的安全通道就可以建立起来。Alice 以发送消息 3 开始，其中包含发给 Bob 的票据，以及一个使用 KDC 刚产生的共享密钥 $K_{A,B}$加密的质询 R_{A2}。然后 Bob 解密该票据以寻找该共享密钥，并返回一个响应 $R_{A2}-1$ 以及对 Alice 的质询 R_B。

下面有关消息 4 的评论是适宜的。一般来说，通过返回 $R_{A2}-1$ 而不只是 R_{A2}，Bob 不

仅证实了他知道该共享密钥，还证实了他确实解密了该质询。此外，这将消息4与消息3联系在一起，与现时RA将消息2和消息1联系在一起的方式相同。因此该协议更有效地防止重放。

然而，在这一特殊情况中，由于此消息之前还没有在该协议中的任何地方使用过这一简单原因，所以返回 $K_{A,B}(R_{A2},R_B)$ 已经足够了。$K_{A,B}(R_{A2},R_B)$ 已经证实了Bob能够解密消息3中发送的质询。图9.17中所示的消息4是由历史原因引起的。

此处介绍的Needham-Schroeder协议仍存在弱点，如果Chuck曾经获得过一个旧密钥 $K_{A,B}$ 的控制，那么他就可以重放消息3，并使Bob建立一个通道。然后Bob将相信他正与Alice会话，虽然实际上另一端是Chuck。在这种情况中，我们需要将消息3与消息1联系起来，也就是说，使该密钥依赖于来自Alice要与Bob建立通道的初始请求。该解决方案如图9.18所示。

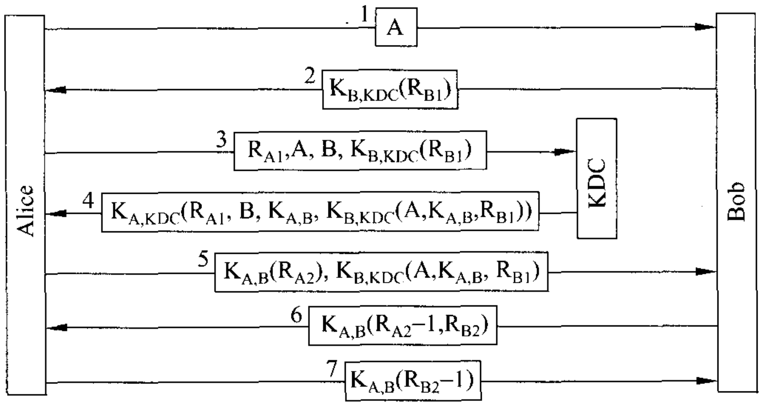


图 9.18 Needham-Schroeder协议中对先前生成的会话密钥恶意地重新使用的防护

其中的窍门是向Alice发送给KDC的请求中加入一个现时。然而，该现时必须来自Bob，这就使Bob确信希望与其建立安全通道的人将从KDC获得适当信息。因此，Alice首先请求Bob向其发送一个使用Bob与KDC之间共享的密钥加密的现时 R_{B1}。Alice将此现时加入到其发往KDC的请求中，随后KDC解密该现时并将结果放在生成的票据中。这样，Bob就会确切知道该会话密钥同Alice要求与Bob交谈的原始请求联系在一起。

9.2.1.3 使用公钥加密的身份认证

现在我们来讨论使用不需要KDC的公钥加密系统的身份认证。我们再一次考虑Alice希望建立到Bob的安全通道，并且两人都拥有彼此的公钥的情况。基于公钥加密的一个典型身份认证协议如图9.19所示，我们接下来要进行解释。

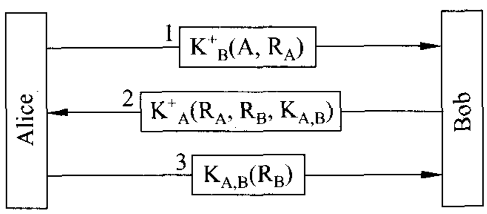


图 9.19 公钥加密系统中的相互身份认证

Alice以向Bob发送一个使用他的公钥 K_B^+ 加密的质询 R_A 开始。解密该消息并向Alice返回该质询是Bob的工作。因为Bob是唯一可以解密该消息（使用与Alice所用的公钥相关联的私钥）的人，所以Alice就会知道她正与Bob交谈。注意，重要的是要保证Alice使用的是Bob的公钥，而不是某个模仿Bob的人的公钥。这种保证方式稍后在本章讨论。

Bob 接收到 Alice 建立通道的请求时，他返回解密过的质询，以及他自身用于对 Alice 进行身份认证的质询 R_B。此外，他还产生一个可以用于更进一步通信的会话密钥 $K_{A,B}$。Bob 对 Alice 的质询的响应、他自己的质询以及该会话密钥都放在一个使用属于 Alice 的公钥 K_A^+ 加密的消息中，如图 9.19 中消息 2 所示。只有 Alice 能够使用与 K_A^+ 相关联的私钥 K_A^- 解密此消息。

最后，Alice 返回其对 Bob 的质询的响应，该响应使用 Bob 产生的会话密钥 $K_{A,B}$。在那种方式下，她就已经证实她能够解密消息 2，因此 Bob 确实是在与 Alice 交谈。

9.2.2 消息的完整性和机密性

除身份认证外，安全通道还应该提供对消息完整性和机密性的保证。消息完整性是指保护消息免受修改；机密性确保了窃听者不能截获和读取消息。机密性可通过在发送消息前简单加密该消息容易地实现。加密通过与接收者共享密钥来实现，或者通过使用接收者的公钥来实现。然而，保护消息免受修改更复杂一些，就像我们接下来要讨论的那样。

9.2.2.1 数字签名

消息完整性通常超出了安全通道中的实际传输的范畴。我们来考虑这样一种情况，即 Bob 刚刚以 500 美元把一个收藏家的一些电唱机唱片卖给 Alice。整个交易是通过电子邮件完成的。最后，Alice 向 Bob 发送了一个消息，确认她将用 500 美元购买那些唱片。除身份认证外，至少需要考虑两个关于消息完整性的问题。

(1) Bob 需要向 Alice 保证他不会恶意地把她的消息中所提的 500 美元改为更高的价格，并声称她答应的价格多于 500 美元；

(2) Alice 需要向 Bob 保证，她不会因为她重新考虑就否认发送过该消息。

如果 Alice 的签名与该消息的内容唯一地联系在一起，那么以这种方式对该消息进行数字签名就可以处理这两个问题。消息和其签名之间的唯一关联防止了对该消息进行了修改而没有引起注意的情况出现。此外，如果可以检验 Alice 的签名是真的，那么她以后就不能否认对该消息进行签名这一事实。

放置数字签名有若干种方式。一种流行的形式是使用例如 RSA 的公钥加密系统，如图 9.20 所示。Alice 向 Bob 发送消息 m 时，她使用其私钥 K_A^- 加密该消息，并向 Bob 发出。如果她还希望保密该消息内容，那么她可以使用 Bob 的公钥并发送 $K_B^+(m, K_A^-(m))$，其中结合了 m 和 Alice 签名的版本。

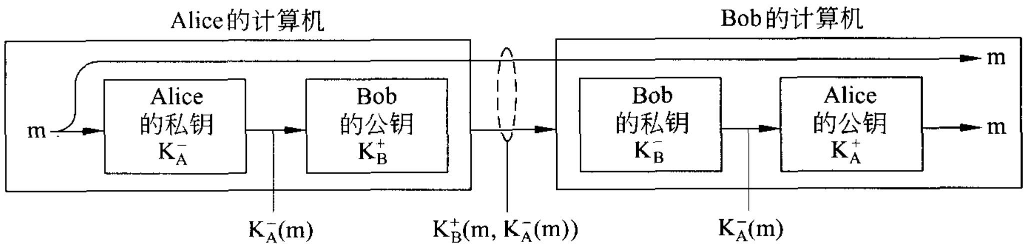


图 9.20 使用公钥加密对消息进行数字签名

该消息到达 Bob 时，他可以使用 Alice 的公钥对其进行解密。如果他能够得到保证该公钥确实属于 Alice，那么解密 m 的签名版本以及成功地与 m 进行比较只能意味着该消息来自 Alice。Alice 受到防止 Bob 对 m 进行任何恶意修改的保护，因为 Bob 必须一直证明 m 的修改版本也是由 Alice 签名的。换句话说，经解密的消息本身在本质上从来不能作为证据。保持 m 的签名版本以保护其自身防止 Alice 的否认也是出于 Bob 自己的利益。

虽然此方法中的协议是正确的，但这一方法仍有很多问题。首先，Alice 的签名的有效性仅在 Alice 的私钥保密的情况下保持。如果 Alice 希望在向 Bob 发送了她的确认后撤销该交易，那么她可以声称她的私钥在该消息发送前被盗了。

Alice 决定改变其私钥时会出现另一个问题。这么做本身可能不是一个坏主意，因为不时改变密钥一般有助于防止入侵。然而，Alice 改变其密钥后，她向 Bob 发送的信息就是无用的了。这种情况下，除了在消息签名时使用时间戳外，还需要跟踪密钥改变时间的一个中央授权机构。

此方法的另一个问题是 Alice 使用其私钥加密整个消息。这样的加密在处理需求上可能开销很大（或者如果我们假设转换为二进制数的消息由一个预定义的最大值限制，那么这种加密在数学上甚至是不能实现的），而且实际上也是不必要的。想想我们仅需要唯一地将签名与特定消息相关联。一个更容易获得且可以证明是更精致的方法是使用消息摘要。

正如我们所解释的，消息摘要是一个固定长度的位串 h，该位串是通过加密散列函数 H 从任意长度的消息 m 计算出来的。如果 m 变为 m'，那么其散列 $H(m')$ 就会与 $h=H(m)$ 不同，这就能够容易地检测出发生了修改。

为了对消息进行数字签名，Alice 可以首先计算一个消息摘要，随后使用其私钥加密该摘要，如图 9.21 所示。经加密的摘要与发给 Bob 的消息一同发送出去。注意，该消息本身作为明文发送：每个人都可以读取该消息。如果需要机密性，那么还应该使用 Bob 的公钥加密该消息。

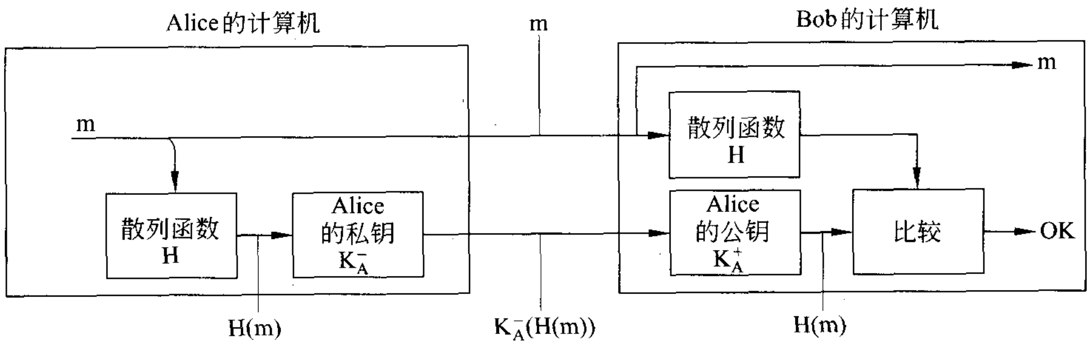


图 9.21 使用消息摘要对消息进行数字签名

Bob 接收到该消息及其经加密的摘要时，他仅需要使用 Alice 的公钥解密该摘要，并单独计算消息摘要。如果从接收到的消息计算出的摘要与经解密的摘要相匹配，那么 Bob 就确信该消息是由 Alice 签名的。

9.2.2.2 会话密钥

在安全通道的建立期间，在身份认证阶段完成以后，通信各方一般使用唯一的共享会话

密钥以实现机密性。而在不再使用该通道时安全地丢弃该会话密钥。还有一种替代办法是为实现机密性使用与建立该安全通道所用密钥相同的密钥。然而,使用会话密钥具有很多重要的好处(Kaufman 等 2003)。

首先,经常使用一个密钥时,就很容易泄露该密钥。在某种意义上,加密密钥就像普通钥匙一样会遭受“磨损”。使用会话密钥的基本思想是如果入侵者能够窃听到许多使用同一密钥加密的数据,就有可能发动攻击以得到所使用的密钥的某些特征,并有可能发现明文或该密钥本身。出于这种原因,尽可能不使用身份认证密钥则更为安全。此外,这样的密钥通常使用一些相对耗时的带外机制进行交换,例如定期的邮件或电话。以这种方式交换密钥应该保持最小量。

为每个安全通道产生一个单独密钥的另一个重要原因是要确保通信各方受到防止重放攻击的保护,正如我们前面碰到过很多次的那样。通过在每次建立安全通道时使用一个唯一的会话密钥,通信各方至少受到防止重放一个完整会话的保护。要保护重放上一个会话中的个别消息,一般需要其他措施,例如加入时间戳或序列号作为消息内容的一部分。

假设消息完整性和机密性是通过使用与建立会话所使用密钥相同的密钥来实现的。在这种情况下,只要该密钥被损坏了,入侵者就可能解密先前对话中传输的消息,这无疑是我们不希望看到的情况。与之相比,每个会话使用一个密钥要安全得多,因为如果这样的一个密钥被损坏了,最坏情况下,只有一个会话受到影响,其他会话期间发送的消息仍可保持机密性。

与这最后一点相关的是 Alice 可能希望与 Bob 交换一些机密数据,但她并不足够信任他,所以她不会以用长期密钥加密的数据的形式向其发送消息。她可能希望为高度机密的消息保留这样的密钥,她与真正信任的人才会交换这些消息。在这样的情况下,使用相对容易得到的会话密钥与 Bob 交谈就已足够。

一般来说,身份认证密钥通常是以这样一种方式建立的,即替换该密钥是要付出相当大的代价。因此,这样的长期密钥与更廉价且临时性更强的会话密钥的结合对实现安全通道的数据交换通常是一个很好的选择。

9.2.3 安全组通信

到目前为止,我们已经集中讨论了在通信双方之间建立安全通信通道的问题。然而,在分布式系统中,通常有必要启用多于两方之间的安全通信。复制的服务器是一个典型的例子,其中所有副本之间的通信都应受到保护,以防修改、伪造和窃听,就像在两方安全通道的情况中一样。在本节中,我们来更彻底地讨论安全组通信。

9.2.3.1 机密组通信

首先,考虑保护 N 个用户的组之间的通信免受窃听的问题。要确保机密性,一个简单的方法是令所有的组成员共享同一密钥,该密钥用于对组成员之间发送的所有消息进行加密和解密。因为此方法中的密钥是所有成员共享的,所以所有成员都应是可靠的,能够确实对该密钥进行保密,这一点很必要。这一先决条件本身就说明,对于机密组通信使用单一共享密钥比两方安全通道更易受到攻击。

一种替代的解决方法是在每对组成员之间使用各自的共享密钥。一旦一个成员被证明在泄露信息，其他成员就会简单地停止向该成员发送消息，但仍然使用他们互相通信所使用的密钥。然而，这样做就不是只保持一个密钥，而是保持 N(N－1)/2 个密钥，这本身就可能是一个困难的问题。

使用公钥加密系统可以解决这个问题。在这种情况下，每个成员有其自己的(公钥，私钥)对，其中所有成员都可用该公钥发送机密消息。在这种情况下，总共需要 N 个密钥对。如果一个成员不再值得信赖，那么仅将其清除出该组，而不会影响其他密钥的使用。

9.2.3.2 安全的复制服务器

现在考虑一个完全不同的问题：一个客户向一组复制服务器发出了一个请求。这些服务器可能已经由于容错或性能的原因进行了复制，但无论如何，该客户都期望响应是可信的。换句话说，不管这组服务器是否像我们在上一章中所讨论的那样经常遭受 Byzantine 故障，客户都希望返回的响应没有受到安全攻击。如果入侵者已经成功地破坏了一台或多台服务器，那么就可能发生这样的攻击。

保护客户免受这样的攻击的一种解决方法是收集来自所有服务器的响应并对每一个服务器进行身份认证。如果响应中的大多数来自未被破坏的(也就是，通过身份认证的)服务器，那么该客户就可以相信该响应也是正确的。不幸的是，这种方法暴露了服务器的复制，从而违反了复制的透明度。

(Reiter 等 1994)提出了安全的复制服务器的方法，其中保持了复制透明度。此方法的优点是，因为客户不知道实际的副本，所以以安全方式添加或删除副本要容易很多。下面讨论密钥管理时我们将回到管理安全组这一话题。

安全和透明的复制服务器的本质在于称为**秘密共享**(secret sharing)的特性。多个用户(或进程)共享一个秘密时，没有人知道整个秘密。相反，该秘密只有在所有人聚集在一起时才能揭开。这样的方法可能非常有用。例如，考虑发射一枚核弹的情形。一般这种行动需要至少两个人的授权。每个人持有一个私钥，该私钥应该与另一个人的私钥结合起来使用才可以实际发射导弹。仅使用一个密钥则不行。

在使用安全的复制服务器的情况下，我们所寻找的是对这样问题的解决方法，即 N 台服务器中最多有 k 台可以产生不正确的答案，且在这 k 台服务器中，最多有 $c \leqslant k$ 台实际上已经被入侵者破坏。注意，这个要求使该服务本身是 k 容错的，如上一章所述。区别在于我们现在将被恶意破坏的服务器归为有故障的一类。

现在考虑服务器被主动复制的情况。换句话说，一个客户将请求同时发送到所有的服务器，随后由每台服务器处理。每台服务器都产生一个返回该客户的响应。对安全复制的服务器组来说，我们要求每台服务器的响应都要附有数字签名。如果 r_i 是来自服务器 S_i 的响应，就令 $md(r_i)$ 表示服务器 S_i 所计算的消息摘要。该摘要使用服务器 S_i 的私钥 K_i^- 签名。

假设我们希望保护客户，防止有多于 c 台的服务器被破坏。换句话说，该服务器组应该能够忍受至多 c 台服务器被破坏，而且还能够产生客户可以给予信任的响应。如果单个服务器的签名能够以这样一种方式结合，即至少需要将 c＋1 个签名联合在一起为该响应构造一个有效的签名，那么这样就可以解决我们的问题。换句话说，我们希望让复制服务器产生

一个保密的有效签名，这样就使在没有至少一台可靠的服务器的帮助下，c 台被破坏的服务器不足以产生有效签名。

作为一个例子，请考虑一个 5 台复制服务器的组，该组应该能够忍受 2 台被破坏的服务器，并且还产生客户可以信任的响应。每台服务器 S_i 向该客户发送其响应 r_i，以及其签名 $sig(S_i, r_i) = K_i^{-}(md(r_i))$。因此，该客户最后会收到 5 个三元组 $\langle r_i, md(r_i), sig(S_i, r_i)\rangle$，从这 5 个三元组中应该得到正确的响应。这一情况如图 9.22 所示。

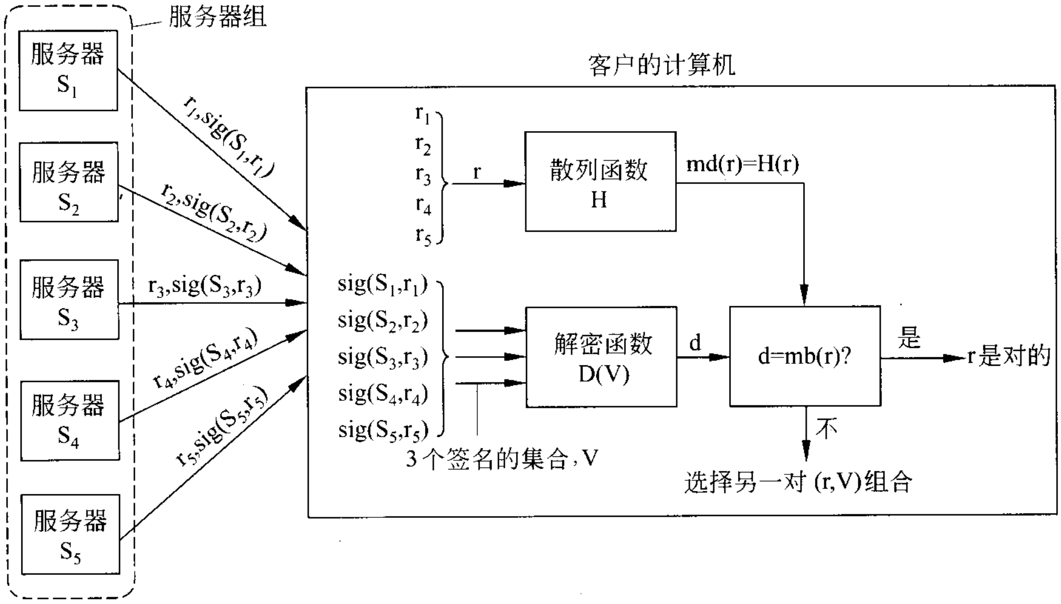


图 9.22 在一个复制服务器组中共享一个保密签名

每个摘要 $md(r_i)$ 也是由客户计算出来的。如果 r_i 不正确，那么通常可以通过计算 $K_i^{+}(K_i^{-}(md(r_i)))$ 检测出来。然而，此方法不再能够应用，因为没有单个服务器值得信赖。相反，客户使用一个特殊的众所周知的解密函数 D，该函数以一个 3 个签名的集合 $V = \{sig(S, r), sig(S', r'), sig(S'', r'')\}$ 作为输入，并制造单个摘要作为输出：

$$d_{out} = D(V) = D(sig(S, r), sig(S', r'), sig(S'', r''))$$

有关 D 的细节，请参阅(Reiter 1994)。3 个签名具有 5! /(3! 2!) = 10 种可能的组合客户可以用它们作 D 的输入。如果其中一种组合为某个响应 r_i 产生了正确的摘要 $md(r_i)$，那么客户就可以认为 r_i 是正确的。特别是，可以相信该响应是由至少 3 台可靠的服务器产生的。

为了提高复制的透明度，Reiter 和 Birman 让每台服务器 S_i 向其他服务器广播一条包含其响应 r_i 以及其相关签名 $sig(S_i, r_i)$ 的消息。一台服务器接收到至少 c+1 个这样的消息(包括其自己的消息)时，该服务器就试图为其中一个响应计算一个有效的签名。如果这对响应 r 和 c+1 个签名的集合 V 成功了，该服务器就以单个消息的形式向客户发送 r 和 V。随后该客户 r 的正确性可以通过检查签名来检验，也就是说，检查 md(r) = D(V) 是否成立。

我们刚刚所描述的内容还被称为**(m,n)-阈值方法**((m,n)threshold scheme)，在我们的例子中，m=c+1 且 n=N，N 为服务器的数量。在一个(m,n)-阈值方法中，一条消息划分为 n 个块，称为**影子**(shadows)，因为任何 m 个影子都可以用于重组原始消息，但使用 m−1 个或更少的影子就不能重组消息。构造(m,n)-阈值方法有多种方式。细节可在文献

(Schneier 1996)中找到。

9.2.4 示例：Kerberos

到如今应该清楚向分布式系统中加入安全性并不是简单的事情。问题是由下列事实引起的，即安全应该是普遍的，否则整个系统的安全性可以轻易受到破坏。为了帮助构建能够实施种种安全策略的分布式系统，已经开发了很多可以作为进一步开发的基础的支持系统。一个广泛使用的重要系统是 **Kerberos**(Steiner 等 1988, Kohl 和 Neuman 1994)。

Kerberos 是在 MIT 开发的，它基于我们早先描述的 Needham-Schroeder 身份认证协议。当前有两个版本的 Kerberos 在使用：版本 4 和版本 5。两个版本概念上很相似，版本 4 解释起来简单很多。版本 5 对版本 4 添加了很多特性，因此一般是首选的。在下文中，我们只集中讨论身份认证问题。

Kerberos 可以看作一个安全系统，该系统帮助客户与服务器建立安全通道。安全是基于共享密钥的。有两个不同组件：**身份认证服务器**(authentication server, AS)和**票据授予服务**(ticket granting service, TGS)。AS 负责处理来自用户的登录请求。AS 对用户进行身份认证并提供一个可以用于与服务器建立安全通道的密钥。建立安全通道是由 TGS 处理的。TGS 分发称为**票据**(ticket)的特殊消息，用于使服务器确信该客户正是其所声称的那个客户。下面我们给出票据的具体例子。

我们来讨论 Alice 登录到使用 Kerberos 的分布式系统的方法，以及她与服务器 Bob 建立安全通道的方法。对 Alice 来说，要登录到该系统，她可以使用任何可用的工作站。该工作站以明文的形式将其名字发送给 AS，AS 返回一个会话密钥 $K_{A,TGS}$ 和一个她将需要移交给 TGS 的票据。

由 AS 返回的票据包含 Alice 的身份，以及 Alice 和 TGS 可以用来互相通信的生成密钥。该票据本身将由 Alice 移交给 TGS。因此，重要的是除 TGS 外没有人能够理解该票据。出于这一原因，该票据使用 AS 和 TGS 之间共享的密钥 $K_{AS,TGS}$ 加密。

登录过程的这一部分如图 9.23 中的消息 1、2 和 3 所示。消息 1 并不是真正的消息，而是对应 Alice 在工作站上键入的她的登录名。消息 2 包含该名称并将其发送到 AS。消息 3 包含会话密钥 $K_{A,TGS}$ 和票据 $K_{AS,TGS}(A, K_{A,TGS})$。为了确保私密性，消息 3 使用 Alice 和 AS 之间共享的密钥 $K_{A,AS}$ 加密。

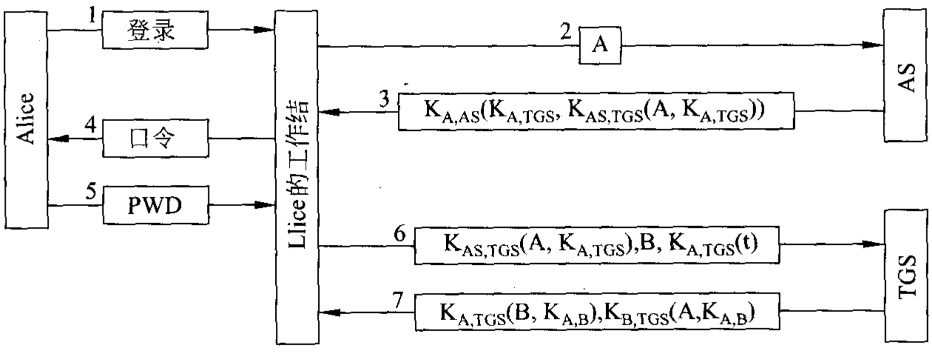


图 9.23 Kerberos 中的身份认证

该工作站接收到来自 AS 的响应时，会提示 Alice 键入其密码(如消息 4 所示)，随后该工作站使用该密码产生共享密钥 $K_{A,AS}$(获得一个字符串密码，应用一个加密散列，然后取前 56 位作为密钥是相对简单的)。注意此方法不只具有 Alice 的密码决不会以明文的形式在网络中发送的优点，而且还有一个优点即该工作站甚至不必临时存储该密码。此外，一旦产生共享密钥 $K_{A,AS}$，该工作站就会找到会话密钥 $K_{A,TGS}$，而且可以忘记 Alice 的密码并只使用共享密钥 $K_{A,AS}$。

身份认证的这一部分进行后，Alice 可以认为其已经登录到该系统中。她现在可以联系其他用户或服务器。如果她想和 Bob 交谈，就请求 TGS 为 Bob 产生一个会话密钥，如图 9.23 中消息 6 所示。Alice 具有票据 $K_{AS,TGS}(A, K_{A,TGS})$这一事实证实了她就是 Alice。TGS 再次使用会话密钥 $K_{A,B}$进行响应，该密钥封装在 Alice 随后必须传送给 Bob 的一个票据中。

消息 6 还包含一个使用 Alice 和 TGS 之间共享的密钥加密的时间戳 t。此时间戳用于防止 Chuck 恶意再次重放消息 6，并试图与 Bob 建立通道。TGS 将在向 Alice 返回一个票据前检验该时间戳。如果它与当前时间的差别大于几分钟，就拒绝对票据的请求。

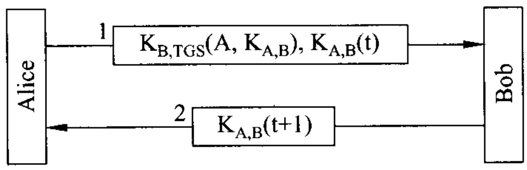


图 9.24 在 Kerberos 中建立安全通道

与 Bob 建立安全通道现在是简单的，如图 9.24 所示。首先，Alice 向 Bob 发送一个消息，其中包含她从 TGS 获得的票据以及一个加密的时间戳。Bob 解密该票据时，会注意到 Alice 将要与其交谈，因为只有 TGS 能够构造该票据。他还找到使其能够检验该时间戳的密钥 $K_{A,B}$。那时，Bob 就确信他在与 Alice 而不是恶意重放消息 1 的某个人交谈。通过使用 $K_{A,B}(t+1)$进行响应，Bob 向 Alice 证实他确实是 Bob。

9.3 访问控制

目前为止在我们所使用的客户-服务器模型中，客户和服务器建立了一条安全通道后，该客户就可以发出要服务器执行的请求。请求包括执行对服务器控制的资源的操作。一种普通情况是一个对象服务器控制很多对象。客户发送的请求一般都包含对特定对象方法的调用。这样的请求只有在客户对该调用具有足够**访问权限**(access right)时才可以执行。

正式地说，检验访问权限称为**访问控制**(access control)，而**授权**(authorization)则是关于授予访问权限的术语。这两个术语互相之间紧密联系，并通常以可互换的方式使用。实现访问控制有很多方式。首先我们讨论一些一般问题，把注意力集中于处理访问控制的不同模型。实际控制对资源的访问的一种重要方式是建立一个防火墙来保护应用程序甚至整个网络。防火墙将被单独讨论。随着代码移动性的出现，访问控制不再能够仅通过使用传统方法来实现，而必须设计新的技术，这一内容也在本节中讨论。

9.3.1 访问控制中的常见问题

为了理解访问控制中所包含的各种问题，通常采用图 9.25 中所示的简单模型。该模型

由发出访问**对象**(object)的请求的**主体**(subject)组成。这里的对象与我们迄今为止所讨论的对象非常相像，可以认为是将自己的状态封装起来并执行对状态的操作。主体可以请求执行的对象操作可以通过接口来使用。最好认为主体是代表用户的进程，但主体也可以是为执行其工作而需要其他对象服务的对象。

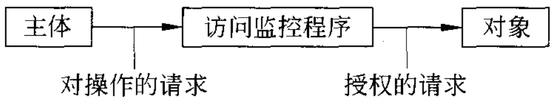


图 9.25 对象访问控制的一般模型

对对象的访问控制与对象保护有关，防止对象被那些不允许其执行特定(或者甚至任何)方法的主体所调用。同样，保护可以包括对象管理问题，例如创建、重命名或删除对象。保护通常由一个名为访问监控器的程序执行。访问监控器记录各个主体可以做的操作，并决定是否允许一个主体执行特定操作。每次调用对象时都调用此监控程序(举例来说，由底层的可信操作系统调用)。因此，该访问监控器本身是可防损害的，这一点非常重要：攻击者一定不能破坏该程序。

9.3.1.1 访问控制矩阵

建立主体对对象的访问权限的模型，其中的一个普通方法是构造一个访问控制矩阵。每个主体由该矩阵中的一行表示，每个对象由一列表示。如果该矩阵由 M 表示，那么一项 M[s,o]就准确地列出了主体 s 可以请求执行的对对象 o 的操作。换句话说，只要主体 s 请求调用对象 o 的方法 m，该访问监控程序就应该检查 m 在 M[s,o]中是否列出。如果 m 没有在 M[s,o]中列出，那么该调用失败。

考虑到一个系统可能需要支持需要保护的上千个用户和上百万个对象，所以不能以真正的矩阵实现访问控制矩阵。矩阵中的许多项将是空的，因为单个主体一般可以访问相对很少的对象。因此，应采用其他更有效的方法来实现访问控制矩阵。

一种广泛应用的方法是让每个对象保存一个希望访问该对象的主体的访问权限表。实质上，这意味着该矩阵以列方向覆盖所有对象分布，不考虑空项。这种类型的实现导致了称为**访问控制列表**(access control list,ACL)。假设每个对象具有其自身的关联 ACL。

另一种方法是以行方向分布该矩阵，通过为每个主体赋予一个其对每个对象所拥有的**权能**(capability)列表来实现。换句话说，一个权能对应访问控制矩阵中的一项。不具备对指定对象的权能意味着没有对该对象的访问权限。

权能可以比喻为票据，赋予其持有者与该票据相关联的特定权限。应该明确的是票据应该受到保护，防止其持有者对其进行修改。一种特别适合分布式系统且广泛应用于 Amoeba(Tanenbaum 等 1990)的方法是使用签名保护(一列)权能。我们稍后在讨论安全管理时再讨论这些内容和其他问题。

使用 ACL 和使用权能这两种保护对一个对象的访问的方式，其区别如图 9.26 所示。在使用 ACL 的情况中，客户向服务器发送一个请求时，服务器的访问监控程序会检查该服务器是否了解该客户以及是否允许该客户执行所请求的操作，如图 9.26(a)所示。

然而，使用权能时，客户仅向服务器发送其请求。该服务器对自己是否了解该客户不感兴趣，权能已经足够了。因此，该服务器仅需要检查该权能是否有效，以及所请求的操作是否在权能表中列出。这种通过权能保护对象的方法如图 9.26(b)所示。

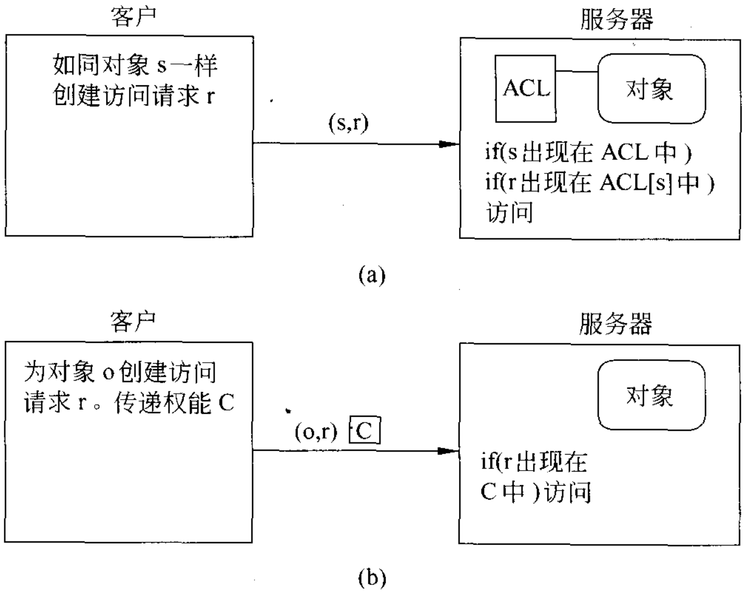


图 9.26　为保护对象而使用的 ACL 和权能之间的比较
(a) 使用 ACL；(b) 使用权能

9.3.1.2　保护域

通过忽略所有的空项，ACL 和权能可以有效地实现访问控制矩阵。尽管如此，如果不采取进一步措施，ACL 或权能列表仍可能变得非常大。

缩小 ACL 的一种普通方式是利用**保护域**(protection domain)。确切地说，保护域是一组(对象，访问权限)对。每对为给定的对象确切指定允许执行的操作(Saltzer 和 Schroeder 1975)。执行一个操作的请求总是在一个域内发出。因此，只要一个主体请求对一个对象执行一个操作，访问监控程序首先都会查找与该请求相关联的保护域。然后，如果找到这个域，那么随后就可以检查是否允许执行该请求。对保护域的使用有几种不同的方法。

其中一种方法是构造用户组(group)。作为例子，我们来考虑一个企业内部网上的一个 Web 页面的情况。这样的一个页面应该对每个雇员都是可用的，但对其他任何人都是不可用的。可能决定令一个单独组 Employee 包含所有当前的雇员，而不是为每个可能的雇员向 ACL 中为该 Web 页面添加一项。无论用户何时访问该 Web 页面，访问监控程序都仅需要检查该用户是否为一个雇员。属于组 Employee 的用户在一个单独的表中记录(当然，该表受到保护，防止未经授权的访问)。

通过引入分层组可以使事情变得更加灵活。例如，如果一个公司拥有三个不同的分公司，分别位于 Amsterdam、纽约和旧金山，那么该公司可能希望将其 Employee 组细分为子组——每个城市一个子组，结果为如图 9.27 所示的一个组织结构。

应该允许所有雇员访问企业内部网的 Web 页面。然而，举例来说，应该只允许 Amsterdam 的雇员子集对与 Amsterdam 分公司相关联的 Web 页面进行修改。如果来自 Amsterdam 的用户 Dick 希望从内部网读取一个 Web 页面，那么访问监控程序首先就需要查找共同组成集合 Employee 的子集 Employee_AMS、Employee_NYC 和 Employee_SF。然后该程序必须检查 Disk 是否包含在这些集合中。分层组的优点是管理组成员相对简单，而且可以有效地构造非常大的组。一个明显的缺点是如果成员数据库是分布式的，那么查

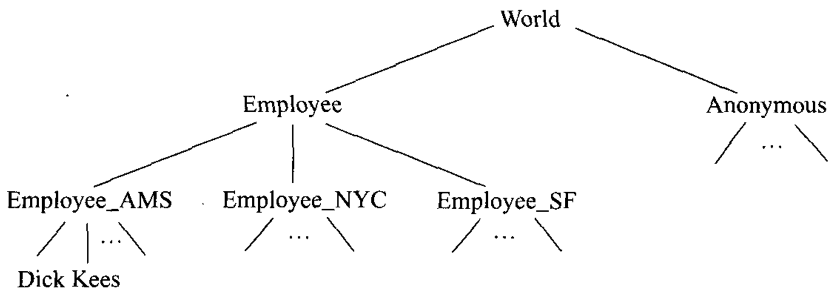


图 9.27 保护域的用户组分层组织

找成员代价可能相当高。

一种替代办法取代了让访问监控程序完成所有工作的方式，该办法是让每个主体携带一个**证书**(certificate)，列出其所属的群组。这样，只要 Dick 希望从企业内部网读取 Web 页面，他就将其证书传送给访问监控程序，声明他是 Employee_AMS 的成员。要保证证书是真实的而且没有被篡改，就应该通过一些手段(例如数字签名)来保护证书。证书看起来与权能差不多。我们稍后再讨论这些问题。

与让组作为保护域相关，还有可能以**角色**(role)来实现保护域。在基于角色的访问控制中，用户始终使用指定的角色登录到系统中，该角色通常与该用户在公司中所具有的职能相关联(Sandhu 1996)。一个用户可以具有多个职能。例如，Dick 可以同时作为部门主管、项目经理和职员调查委员会的成员。根据他登录时所采用的角色，可以分配给他不同的特权。换句话说，他的角色决定了他所在的保护域(也就是组)。

当为用户分配角色并要求用户在登录时采用指定的角色时，用户还应该可以在必要时改变其角色。例如，可能需要允许作为部门主管的 Dick 偶尔改变为项目经理。注意，这样的改变在仅以组来实现保护域时是难以表示的。

除使用保护域外，还可以通过基于对象所提供的操作将对象(分层地)分组来进一步提高效率。例如，根据对象提供的接口对对象进行分组，而不是考虑独立的对象，可能使用子类型(也称为接口继承，参见 Gamma 等 1994)来实现分层。在这种情况中，一个主体请求对一个对象执行操作时，访问监控程序查找该对象的这个操作所属的接口。然后该程序检查是否允许该主体调用属于该接口的操作，而不是检查该主体是否能调用指定对象的操作。

将保护域和对象分组结合起来也是可能的。使用这两种技术，以及指定的数据结构和有限的对象操作，(Gladney 1997)描述了如何为在数字图书馆中使用的非常大的对象集合实现 ACL 方法。

9.3.2 防火墙

到目前为止，我们已经说明了使用加密技术，结合访问控制矩阵的一些实现建立保护的方法。只要所有的通信方根据相同的规则集工作，这些方法就能发挥作用。当开发与世界其余部分隔离的独立分布式系统时，这样的规则可能还要进一步强化。然而，允许外来人访问分布式系统控制的资源时，事情就变得更为复杂。这种访问的例子包括发送邮件、下载文件、上载传真表格等。

为了在这些环境下保护资源，就需要一种不同的方法。实际上，所发生的对分布式系统任何部分的外部访问都由一种称为防火墙（firewall）（Cheswick 和 Bellovin 2000；Zwicky 等 2000）的特殊类型的访问监控程序控制。本质上，防火墙将分布式系统的任意部分与外界分离开，如图 9.28 所示。所有出入站数据包，尤其是所有入站数据包都通过一台特殊计算机传送，并且在传送前受到检查。未经授权的通信数据将被丢弃，不允许继续传输。一个重要问题是防火墙本身应该受到很好的保护，以防受到任何类型的安全威胁，它决不应该出故障。

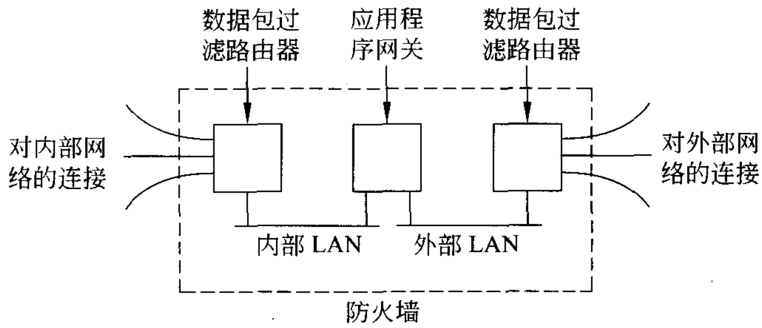


图 9.28 防火墙的一般实现方法

防火墙在本质上分为两种不同的类型，通常结合在一起使用。防火墙的一个重要类型是**数据包过滤网关**（packet-filtering gateway）。这种类型的防火墙作为路由器工作，并基于数据包报头中包含的源地址和目的地址制定关于是否传送一个网络数据包的决定。通常，图 9.28 中所示的外部 LAN 上的数据包过滤网关会过滤入站数据包，而内部 LAN 中的网关则会过滤出站数据包。

例如，为了保护一个内部 Web 服务器，防止来自不属于内部网络的主机的请求进入，数据包过滤网关可以决定丢掉所有发往该 Web 服务器的入站数据包。

更微妙的是这种情况，例如，公司的网络由多个局域网组成，通过我们前面所讨论的 SMDS 网络相连接。每个 LAN 都可通过一个数据包过滤网关进行保护，该网关配置为仅让源于一个其他 LAN 上的主机发送的入站数据通过。一个专用虚拟网络可以用这种方式建立。

另一种类型的防火墙是**应用层网关**（application-level gateway）。与仅检查网络数据包报头的数据包过滤网关相比，这种类型的防火墙实际检查入站或出站消息的内容。一个典型实例是邮件网关，该网关丢弃超过一定大小的入站或出站邮件。也存在更复杂的邮件网关，例如，能够过滤垃圾电子邮件的网关。

应用层网关的另一个例子是允许对数字图书馆服务器进行外部访问，但仅提供文档摘要的网关。如果外部用户想获得更多内容，就要启动一个电子付费协议。防火墙内的用户可以直接访问该图书馆服务。

应用层网关的一种特殊类型称为**代理网关**（proxy gateway）。这种类型的防火墙作为指定类型的应用程序的前端工作，并且确保只传送符合某个准则的消息。作为实例，我们来考虑在 Web 上冲浪的问题。正如我们在下一节要讨论的，许多 Web 页面包含要在用户浏览器中执行的脚本或小应用程序。为了防止将这样的代码下载到内部 LAN 中，所有 Web 通信都应直接通过 Web 代理网关。该网关接受正常的 HTTP 请求，不论是来自防火墙内

部还是外部。换句话说，该网关对用户表现为一个一般的 Web 服务器。然而，它过滤所有的入站和出站通信信息，或者丢弃某些请求和页面，或者在页面中包含可执行代码时修改页面。

9.3.3 安全的移动代码

正如我们在第 3 章中所讨论的，现代分布式系统中的一个重要发展是在主机之间移植代码的能力，而不是仅仅迁移被动数据的能力。然而，移植代码带来了很多严重的安全威胁。首先，在因特网上发送一个代理程序时，其所有者将希望保护该代理程序，防止恶意主机盗窃或修改该代理程序所带的信息。

另一个问题是需要保护主机防止恶意代理程序的破坏。大多数分布式系统的用户都不是系统技术的专家，他们不知道是否可以信任他们从另一台主机获得的程序不会破坏其计算机。在许多情况下，即使对专家来说完全检测一个正在被实际下载的程序也可能是很困难的。

除非采取安全措施，否则恶意程序进入计算机之后就可以轻易地破坏其宿主机。我们面临访问控制问题：不应该允许程序对主机资源未经授权的访问。正如我们要看到的，保护主机防止下载的恶意程序的破坏并不总是容易实现的。问题并不是要避免程序下载。相反，我们所寻求的是对移动代码的支持，允许以灵活且完全可控制的方式访问本地资源。

9.3.3.1 保护代理

在我们讨论保护计算机系统免受下载的恶意代码的破坏之前，首先来看相反的情况。我们来考虑一个移动代理程序，该代理程序以一个用户的名义在一个分布式系统中漫游。这样的一个代理程序可能正在搜索从 Nairobi 到 Malindi 最便宜的机票，并且已由其所有者授权，一旦找到一架班机就预订票。出于这一目的，该代理程序可以携带一张电子信用卡。

显然，在这里我们需要保护。当该代理程序移动到一台主机时，不应该允许该主机窃取该代理程序的信用卡信息。同样，该代理程序也应受到保护以防会使所有者支付比实际需要更多的金额的修改。例如，如果 Chuck 的 Cheaper Charters（便宜机票契约商）可以看到该代理程序还没有访问过价钱比其更便宜的竞争者 Alice Airlines，那么他就会防止 Chuck 改变该代理程序使其不会访问 Alice Airlines 的主机。需要保护代理程序防止来自敌对主机的攻击的另外一些例子包括恶意破坏代理程序，或篡改代理程序以便在其返回时进行攻击或从其所有者处窃取信息等。

不幸的是，不可能完全保护代理程序免受所有类型的攻击（Farmer 等 1996）。这种不可能主要是由无法确实保证主机将完成其许诺过的事情这一事实引起的。因此一种替代方法是以至少可以检测出修改的方式组织代理程序。在 Ajanta 系统中遵循了这种方法（Karnik 和 Tripathi 2001）。Ajanta 提供了三种机制，允许代理程序所有者检测该代理是否被篡改，即只读状态、只追加记录和有选择地揭示某些服务器的状态。

Ajanta 代理程序的**只读状态**（read-only state）由代理程序所有者签名的数据项集合组成。签名在代理程序发送到其他主机前构建和初始化时发生。所有者首先构建一个消息摘要，然后使用其私钥加密该摘要。当该代理程序到达一台主机时，该主机能够通过与原始状

态的签名消息摘要比较检验它的状态，可以很容易检测出只读状态是否已被篡改。

为了允许代理程序在主机间移动的过程中能够收集信息，Ajanta 提供了安全的**只追加日志**(append-only log)。这些日志的特征是数据只能追加到日志后面，在所有者不能检测的情况下不能删除或修改数据。使用只追加日志工作方式如下。最初，该日志是空的，并只有一个以 $C_{init}=K_{owner}^{+}(N)$ 计算的关联校验和 C_{init}，其中 K_{owner}^{+} 是该代理所有者的公钥，N 是只有所有者知道的一个秘密**现时**(nonce)。

当该代理程序移至希望传给其一些数据 X 的服务器 S 时，S 将 X 追加到日志后，然后使用其签名 sig (S,X)签署 X，并计算出校验和：

$$C_{new}=K_{owner}^{+}(C_{old},sig(S,X),S)$$

其中 C_{old} 是以前使用的校验和。

当该代理程序回到其所有者处时，所有者可以容易地检验日志是否已被篡改。所有者从结尾开始读取记录，接连对校验和 C 计算 $K_{owner}^{-}(C)$。每次迭代都为下一次迭代返回一个校验和 C_{next}，以及某个服务器 S 的 sig (S,X)和的 S。然后所有者就可以检验记录中的 then-last 元素是否与 sig(S,X)相匹配。如果匹配，就删除该元素并进行处理，然后进行下一次迭代步骤。当到达初始校验和，或因为签名不匹配使所有者注意到记录已遭篡改时停止迭代。

最后，Ajanta 支持有**选择地揭示状态**(selective revealing)，该机制通过提供一个数据项矩阵来实现，其中每项为一个指定的服务器。每项都使用指定的服务器的公钥进行加密以确保机密性。整个矩阵由该代理的所有者签名以确保该矩阵作为整体的完整性。换句话说，如果该矩阵的任一项被恶意主机修改，那么任何指定的服务器都会注意到并采取适当行动。

除保护代理防止恶意主机的破坏外，Ajanta 还提供各种机制来保护主机防止恶意代理的破坏。正如我们接下来要讨论的，许多这种机制还由支持移动代码的其他系统所提供。有关 Ajanta 的进一步信息可在(Tripathi 等 1999)中找到。

9.3.3.2 保护目标

虽然保护移动代码防止恶意主机的破坏是重要的，但人们对保护主机防止恶意移动代码的破坏则给予了更多的注意。如果认为将代理程序发往外部世界太危险，那么用户一般可选择自己完成代理所做的工作。然而，除了完全封锁，通常没有可供选择的方法使代理程序进入用户的系统。因此，如果决定代理程序可以进入系统，那么用户就需要对该代理程序所能做的操作进行完全控制。

正如我们刚刚讨论过的，虽然保护代理程序免遭修改也许不可能，但至少代理程序所有者可能检测到发生了修改。在最坏情况下，所有者将不得不在代理程序返回时丢弃该代理程序，只有这样才不会造成任何损害。然而，处理恶意入站代理程序时，仅检测您的资源是否受到攻击已经太晚了。相反，保护所有资源免受已下载代码的未经授权的访问是基本的功能。

一种保护方法是构造一个沙箱。**沙箱**(sandbox)是一种技术，使用这种技术下载的程序可以这样的方式执行，即它的每条指令

都能够完全被控制。如果某个程序试图执行主机禁止的一条指令，那么该程序的执行

就会终止。同样，指令访问主机不允许访问的某些寄存器或内存空间时该程序的执行也会终止。

实现沙箱并不容易。一种方法是在下载可执行代码时对其进行检查，并对只能在运行时检查的情况插入其他指令（Wahbe 等 1993）。幸运的是，在处理解释的代码时，事情变得简单很多。我们来简要地考虑 Java 中所采用的方法（参阅（MacGregor 等 1998））。每个 Java 程序都由很多创建对象的类组成。没有全局变量和函数；所有内容都必须声明为一个类的一部分。程序执行从一个名为 main 的方法开始。一个 Java 程序编译为一组由称为 **Java 虚拟机**（Java Virtual Machine）解释的指令。对一个客户来说，要下载和执行编译过的 Java 程序，该客户进程必须运行 JVM。JVM 随后将从组成 main 方法的指令开始，通过解释每条指令来处理下载程序的实际执行。

在一个 Java 沙箱中，保护是通过确保处理将程序传输到客户机器的组件是可信的开始的。Java 中的下载由一组**类加载程序**（class loader）负责。一个类加载程序负责从服务器上获取指定的类，并将其安装在客户的地址空间，以使 JVM 能够从中创建对象。类加载程序只是另一个 Java 类，所以一个下载程序有可能包含其自己的类加载程序。沙箱所处理的第一件事是仅使用可信的类加载程序。特别是，不允许 Java 程序创建自己的类加载程序，通过该加载程序可以不使用类加载通常的处理方式。

Java 沙箱的另一个组件由一个**字节代码验证器**（byte lode verifier）组成，该验证器检查下载的类是否服从该沙箱的安全规则。特别是，该验证器检查该类是否包含不合法的指令或可能以某种方式破坏堆栈或内存的指令。并不是所有的类都要检查，如图 9.29 所示，需要检查的仅仅是从外部服务器下载到客户的类。客户机器上的类一般是可信的，虽然其完整性也可以很容易地进行检验。

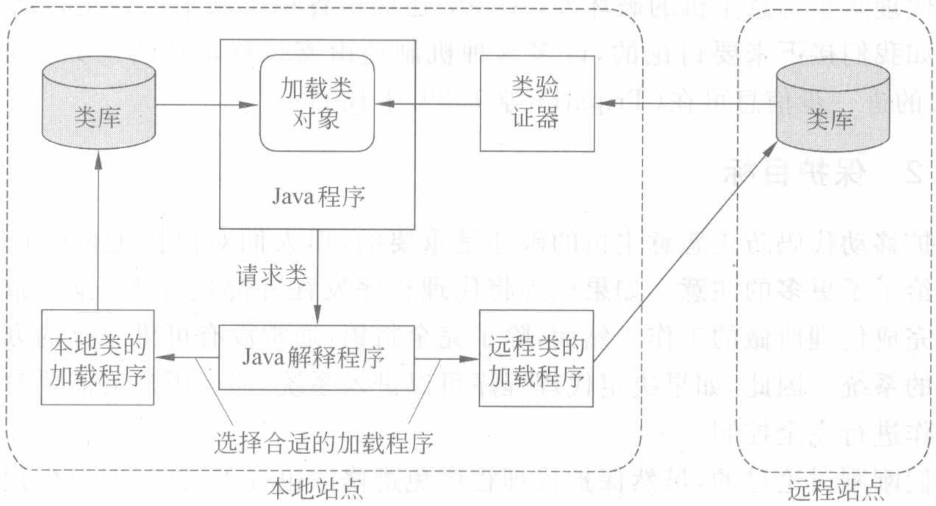


图 9.29 Java 沙箱的组织结构

最后，一个类安全地下载下来并通过检验后，JVM 就可以从中实例化对象并执行这些对象的方法。要进一步防止对象对客户资源未经授权的访问，就应使用在运行时执行各种检查的**安全管理程序**（security manager）。要下载的 Java 程序被迫使用安全管理程序，没有任何方法可使其避免对该程序的使用。这意味着，例如，任何 I/O 操作都受到检查以确定有效性，如果安全管理程序说“不”，就不能执行该操作。因此安全管理程序扮演我们前面

讨论的访问监控程序的角色。

典型的安全管理程序禁止执行许多操作。例如，事实上所有安全管理程序都拒绝访问本地文件，且仅允许程序建立到其所在地服务器的连接。显然也不允许使用JVM。然而，允许程序访问图形库以进行显示，并可以捕获诸如鼠标移动或单击鼠标按键这样的事件。

最初的Java安全管理程序实现了相当严格的安全策略，该策略中不同下载程序之间或甚至来自不同服务器的程序之间没有区别。在许多情况中，最初的Java沙箱模型限制太严格，需要更多的灵活性。下面，我们讨论当前所采用的一种替代方法。

符合沙箱但提供稍多灵活性的一种方法是为下载的移动代码创建运动场(Malkhi和Reiter 2000)。**运动场**(playground)是一台专门为运行移动代码保留的单独的指定机器。在运动场内执行的程序可以使用运动场本地的资源，例如文件或到外部服务器的网络连接，这些程序服从标准保护机制的管理。然而，其他机器本地的资源物理上与运动场分离，下载的代码不能访问到这些资源。这些其他机器上的用户可以用传统方式访问该运动场，例如，通过RPC。然而，没有任何移动代码可以下载到运动场外的机器上。沙箱和运动场的区别如图9.30所示。

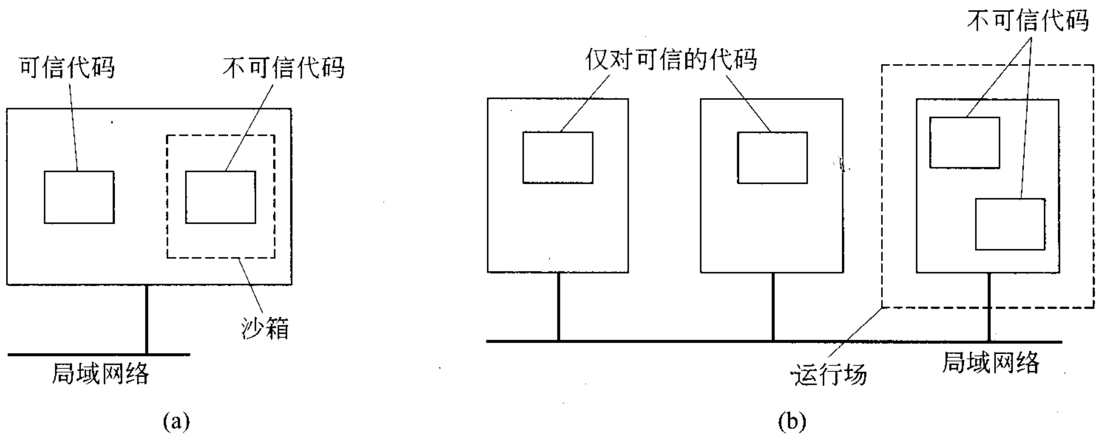


图9.30 沙箱和运动场
(a) 沙箱；(b) 运动场

趋向更强灵活性的下一步是要求每个下载的程序能够通过身份认证，随后基于该程序的来源执行指定的安全策略。要求程序能够通过身份认证相对容易：可以对移动代码签名，正如对任何其他文档一样。这种代码签名方法通常还作为沙箱的替代方法使用。实际上，只有来自可信服务器的代码才被接受。

然而，困难部分是实施安全策略。(Wallach等1997)对Java程序提出了三种机制。第一种方法使用基于作为权能的对象引用。要访问诸如文件一类的本地资源，程序必须在下载前已经获得对处理文件操作的指定对象的引用。如果没有获得这种引用，就不可能访问文件。此原则如图9.31所示。

所有实现文件系统的对象的接口最初都对程序隐藏，这是通过简单地不向这些接口分发任何引用来实现的。Java的强类型检查确保了在运行时建立一个这种接口的引用是不可能的。此外，我们可以使用Java的属性来保存完全属于一个类内部的某些变量和方法。特别是，可以防止一个程序实例化其自己的文件处理对象，这可以通过从本质上隐藏由给定类创建新对象的操作来实现。在Java术语中，构造函数对其关联的类是私有的。

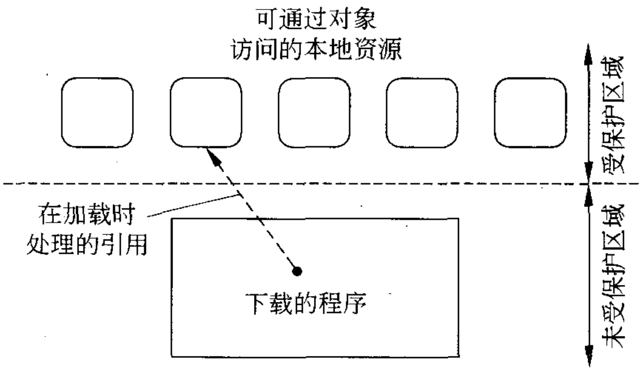


图 9.31 使用作为权能的 Java 对象引用的原则

执行安全策略的第二种机制是(**扩展的**)**栈自省**((extended)stack in trospection)。本质上，对一个本地资源的方法 m 的任何调用都在一个对特殊过程 enable_privilege 的调用之后，该过程检查是否把对该资源调用 m 方法的权限授予调用者。如果该调用是经过授权的，那么该调用者就获得临时特权，持续时间为该调用期间。m 完成时，在将控制返回给调用者之前，调用特殊过程 disable_privilege 来禁用这些特权。

为了执行对 enable_privilege 和 disable_privilege 的调用，要求本地资源接口的开发人员将这些调用插入在适当的位置。然而，令 Java 解释器自动处理这些调用要好很多。举例来说，这是用于处理 Java 小应用程序的 Web 浏览器所遵循的标准方法。一个精致的解决方法如下：只要进行本地资源的一次调用，Java 解释器就会自动调用 enable_privilege，随后该过程检查是否允许进行该调用。如果允许，就将对 disable_privilege 的调用推入栈中以确保该方法调用返回时禁用特权。这种方法防止了恶意编程人员以欺诈回避规则。

使用栈的另一个重要优点是其启用了一种更好的方式来检查特权。假设一个程序调用一个本地对象 O1，该对象又调用对象 O2。虽然 O1 可能具有调用 O2 的权限，但如果不能信任 O1 的调用者调用属于 O2 的一个指定方法，那么就不应该允许这一链式调用。栈自省使检查这样的链很容易，因为解释器仅需要检查每个位于栈顶端的帧来了解是否存在一个帧具有已启用的正确特权（这种情况下允许调用），或者是否存在一个明确禁止对当前资源访问的帧（这种情况下调用被立即终止）。此方法如图 9.32 所示。

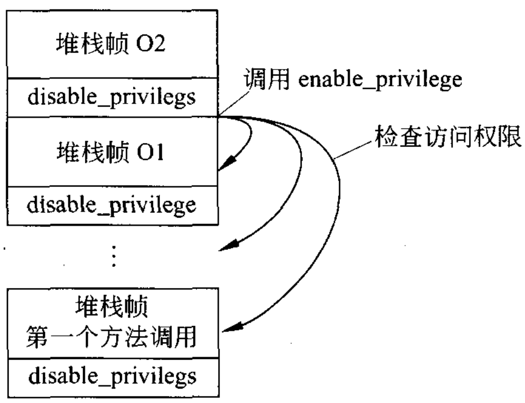


图 9.32 堆栈自省的原理

本质上，栈自省允许对类或方法的特权进行连接，并对每个调用者的那些特权的分别检查。这样，就可能实现基于类的保护域，在(Gong 和 Schemers 1998)中进行了详细解释。

实现安全策略的第三种方法是通过**名称空间管理**(name space management)。其主导思想如下：为了赋予程序对本地资源的访问权限，首先需要通过加入包含实现那些资源的类的适当文件来获得访问。加入过程要求赋予解释器一个名称，然后将该名称解析为一个类，随后在运行时加载该类。要为指定的下载程序实施安全策略，可以根据下载程序的来源将同一名称解析为不同的类。通常，名称解析由类加载程序处理，该程序需要进行修改以实

现这种方法。有关细节可在(Wallach 等 1997)中找到。

到目前为止我们所描述的是基于下载程序的来源将特权与类或方法关联起来的方案。依靠 Java 解释器,通过上面所描述的机制执行安全策略是可能实现的。在这种意义上,安全结构变得高度依赖语言,并且需要为其他语言重新开发。独立于语言的解决方案,例如在(Jaeger 等 1999)中所描述的,需要一种更普通的方法来加强安全,并且还更难以实现。在这些情况下,需要安全操作系统的支持,该系统意识到下载移动代码,并且强迫所有对本地资源的调用在随后进行检查的内核中运行。

9.3.4 拒绝服务

通常,访问控制是确保只有经授权的进程蔡可以访问资源。与访问控制有关的一个非常麻烦的攻击是恶意地阻止已授权进程访问资源。随着分布式系统经由 Internet 的开放,防止**拒绝服务攻击**(denial-of-service(DoS) attack)变得越来越重要了。对付来自一个或多个资源的 DoS 往往很高效,但要对付**分布式拒绝服务**(distributed denial of service,DDoS)就变得困难多了。

在 DDoS 攻击中,大量的进程一起试图摧垮一个网络服务。在这些情况下,我们往往可以看到,攻击者已经成功地劫持了一大组机器,而这些机器毫无意识到已经加入到攻击的行列中了。(Specht 和 Lee 2004)把攻击分成了两类:带宽耗竭的攻击和资源耗竭的攻击。

带宽耗竭是往某个机器发送大量消息。结果是,正常的消息很难到达接收者。资源耗竭攻击主要是使接收者把资源消耗在无用的消息上。一个著名的资源耗竭攻击是 TCP SYN 泛洪。在这种情况下,攻击者试图初始化大量的连接(即,作为第三方握手的一部分发送 SYN 数据包),但并不响应来自接收者的确认消息。

目前并没有哪个方法能防止 DDoS 攻击。一个问题是,攻击者会利用无知的受害者,在他们机器上秘密地安装软件。在这种情况下,唯一的解决办法是,让机器通过检查所安装的文件,不停地监视它们的状态。病毒可以在因特网上扩散,如果仅仅依靠反病毒措施是不可行的。

更好的办法是不断地监视网络流量,例如,在出口路由器(数据包就是从这里离开企业网络的)处进行监视。经验表明,把源地址不属于任何企业网络的数据包删除掉,可以防止很多攻击破坏。通常,最好是把数据包严格地按源地址过滤。

另外,也可以在入口路由器进行监视,也就是说,在数据流入企业网络时进行监视。问题是,在入口路由器检测攻击可能会太晚了,因为正常的数据有可能已经无法达到该网络了。最好是让路由器远离因特网,如在 ISP 的网络中,当怀疑有攻击要开始时就删除数据包。这种方法是由(Gil 和 Poletto 2001)提出的,当路由器发现到某个结点的数据包数目与来自该结点的数据包数目比例不对称时,就删除掉这些数据包。

通常,需要部署不同的反攻击技术,因为会有新的攻击方法不断出现。(Mirkovic 等 2005)介绍了拒绝服务攻击和解决办法,(Mirkovic 和 Reiher 2004)介绍了攻击的详细分类。

9.4 安全管理

到目前为止，我们已经考虑过安全通道和访问控制，但几乎没有谈到安全管理的问题，例如密钥的获得方法。在本节中，我们更深入地讨论安全管理。特别是，我们分成三个主题进行讨论。首先，我们需要考虑加密密钥的一般管理，尤其是公钥的分配方式。正如所证明的，证书在这里起重要作用。

其次，我们通过集中讨论添加当前成员信任的一个新的群组成员来讨论安全管理一组服务器的问题。很明显，面对分布式的和复制的服务，即使接纳一个恶意进程到一个组也不要危及安全这一点是重要的。

第三，我们通过考虑权能和称为属性证书的事物来关注授权管理。分布式系统中关于授权管理的一个重要问题是一个进程可以将它的一些或所有访问权限委派给另一个进程。以安全方式委派权限具有其自身的微妙之处，正如我们在本节中所讨论的。

9.4.1 密钥管理

到目前为止，我们已经描述了各种加密协议，其中我们（隐含地）假设各种密钥是容易获得的。例如，在公钥加密系统的案例中，我们假设一条消息的发送者可以使用接收者的公钥，以使其能够加密该消息来确保机密性。同样，在使用密钥分发中心（KDC）的情况下，我们假设每一方都已经与 KDC 共享一个密钥。

然而，建立和分发密钥并不是微不足道的事情。例如，通过不安全的通道分发密钥是不可接受的，在许多情况下我们需要采取带外方法。同样，也需要一些机制来废止密钥，也就是说，要防止密钥受到损害或无效后被继续使用。例如，废止受到损害的密钥是必要的。

9.4.1.1 密钥建立

我们首先来考虑会话密钥的建立方式。Alice 希望与 Bob 建立一个安全通道时，她可能首先使用 Bob 的公钥来开始通信，如图 9.19 所示。如果 Bob 接受，他随后就可以生成会话密钥并将其用 Alice 的公钥加密后返回给 Alice。通过在传输前加密共享会话密钥，该密钥就可以安全地在网络中传送。

一种相似的方法可以用于在 Alice 和 Bob 已经共享一个密钥时生成并分发会话密钥。然而，这两种方法都要求通信各方已经具有可用的方法来建立安全通道。换句话说，必须已经发生了一些形式的密钥建立和分发。共享密钥通过诸如 KDC 这样的可信的第三方建立时相同的理论也适用。

在不安全的通道中建立共享密钥的一种精致且广泛应用的方法是 **Diffie-Hellman 密钥交换**（Diffie-Hellman key exchange）（Diffie 和 Hellman 1976）。该协议工作方式如下：假设 Alice 和 Bob 希望建立一个共享密钥。第一要求就是他们约定两个比较大的数 n 和 g，这两个数应具有一定的数学属性（我们在这里不予讨论）。n 和 g 两者都设为公用的，没有必要对外界人员隐藏。Alice 选择一个比较大的随机数 x，她将其保密。同样，Bob 也选择他自

己的保密的比较大的数 y。此时就有足够的信息来构造密钥，如图 9.33 所示。

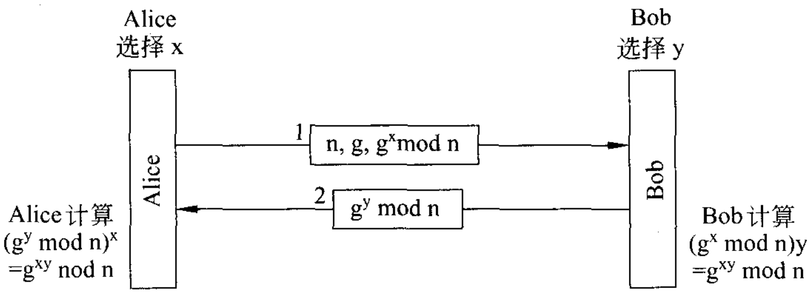


图 9.33 Diffie-Hellman 密钥交换的原理

Alice 以向 Bob 发送 g^x mod n 以及 n 和 g 开始。注意此信息可以明文发送这一点很重要，因为事实上不可能在给定 g^x mod n 的条件下计算出 x。Bob 接收到该消息时，他随后计算(g^x mod n)y，相当于 g^{xy} mod n。此外，他向 Alice 发送 g^y mod n，然后 Alice 可以计算(g^y mod n)^x = g^{xy} mod n。因此，Alice 和 Bob 两人，而且只有他们两人，此刻就具有共享密钥 g^{xy}mod n。注意他们都不需要让对方知道其私有数(分别为 x 和 y)。

Diffie-Hellman 可以看作一个公钥加密系统。对于 Alice 来说，x 是她的私钥，而 g^x mod n 则是她的公钥。正如我们接下来要讨论的，安全地分发公钥对于使 Diffie-Hellman 能实际工作是最基本的。

9.4.1.2 密钥分发

密钥管理中一个更困难的部分是初始密钥的实际分发。在对称加密系统中，初始共享密钥必须沿着一条提供身份认证和机密性的安全通道交流，如图 9.34(a)所示。如果 Alice 和 Bob 没有可用的密钥建立这样的安全通道，就有必要在带外分发该密钥。换句话说，Alice 和 Bob 必须使用一些其他通信手段而不是通过网络互相联系。例如，他们中的一人可以给另一人打电话，或者使用缓慢的邮寄方式用软盘发送密钥。

在使用公钥加密系统的情况下，我们需要以这样一种方式分发公钥，即接收者能够确信该密钥肯定可以与声明的一个私钥配成对。换句话说，如图 9.34(b)所示，虽然公钥本身可以用明文发送，但是发送该公钥所通过的通道必须可以提供身份认证。当然，私钥也需要通过提供身份认证和机密性的安全通道发送。

谈到密钥分发时，通过身份认证的公钥分发可能是最有意思的。实际上，公钥分发是通过使用**公钥证书**(public-key certificate)进行的。这样的一个证书由一个公钥与一个标识该公钥所关联的实体的字符串共同组成。该实体可以是一个用户，也可以是一台主机或某个特殊设备。该公钥和标识符共同由**认证机构**(certification authority)签发，并且此签名也置于该证书上。(认证机构的身份自然是证书的一部分。)签名通过一个属于该认证机构的私钥 K_{CA}^-进行。相应的公钥 K_{CA}^+被认为是众所周知的。例如，各种认证机构的公钥嵌入到大多数 Web 浏览器内部并用二进制数装载。

使用公钥证书的工作方式如下。假设一个客户希望确定在证书中找到的公钥确实属于已标识的实体。然后该客户使用关联的认证机构的公钥来检验该证书的签名。如果证书上的签名与(公钥，标识符)对相匹配，那么该客户就确认该公钥确实属于该已标识的实体。

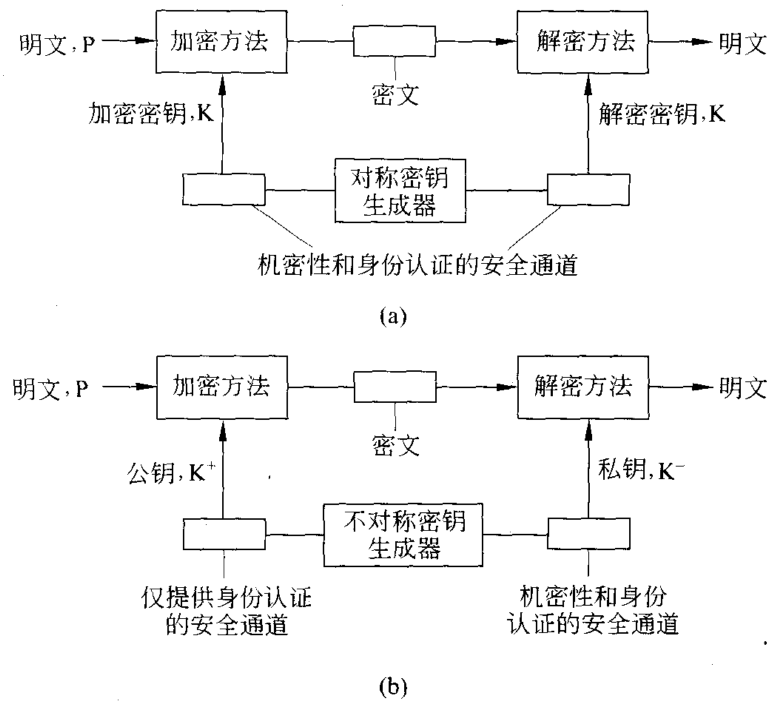


图 9.34 密钥分发和公钥分发（参见文献 Menezes 等 1996）
(a) 密钥分发；(b) 公钥分发

重要的一点是，注意通过确认证书是处于可用状态的，客户实际上就相信该证书不是伪造的。特别是，客户必须假设公钥 K_{CA}^{+} 确实属于关联的证书授权机构。如果有所怀疑，就应该能够通过来自可能更可靠的不同认证机构的另一个证书来检验 K_{CA}^{+} 的有效性。

必须每个人都信任最高级的证书颁发机构这样的分层**信任模型**（trust model）并不是罕见的。例如，**增强型私密电子邮件**（privacy enhanced mail，PEM）使用一个三级信任模型，其中最低级的证书颁发机构可以由**策略认证机构**（policy certification authorities，PCA）进行身份认证，而 PCA 由**因特网策略注册机构**（Internet policy registration authority，IPRA）进行身份认证。如果一个用户不信任 IPRA，或者不认为他可以安全地与 IPRA 谈话，那么他就永远不可能信任使用 PEM 时电子邮件消息能够以安全方式发送。有关此模型的详细信息可在（Kent 1993）中找到。其他信任模型在（Menezes 等 1996）中予以讨论。

9.4.1.3 证书的生存期

有关证书的一个重要问题是其寿命。首先我们来考虑一下证书颁发机构分发终生证书的情况。本质上，该证书声明的是该公钥对该证书所标识的实体将一直有效。显然，这样的声明并不是我们想要的。如果被标识的实体的私钥曾经受到损害，就没有可信的客户会永远能够使用该公钥（更不用说恶意客户）。在这种情况下，我们需要一种机制，通过令公众知道该证书不再有效来吊销该证书。

吊销证书有多种方式。一种普通方法是使用证书颁发机构定期公布的**证书吊销表**（certificate revocation list，CRL）。一个客户无论何时检查一个证书，都必须检查 CRL 来查看该证书是否已经被吊销。这意味着该客户至少必须在每次公布新的 CRL 时联系证书颁发机构。注意，如果 CRL 是每天公布的，那么吊销证书也需要一天。同时，一个已损坏的证

书可能被错误地使用，直到在下一期 CRL 中公布。因此，公布 CRL 之间的时间不能太长。此外。获得一个 CRL 会导致一些开销。

一种替代方法是限制证书的生存期。实质上，此方法与我们在第 6 章中所讨论的租用分发相似。证书的有效性在一段时间后自动终止。如果出于任何原因要在到期前吊销证书，那么证书颁发机构依然可以在 CRL 上公布该证书。然而，此方法仍会迫使客户只要检验证书就要检查最新的 CRL。换句话说，客户需要联系证书颁发机构或包含最新 CRL 的可靠数据库。

最后的一种情况是将证书的生存期缩短到几乎为零。实际上，这就意味着不再使用证书；作为替代，客户必须一直联系证书颁发机构以检查公钥的有效性。结果是，该证书颁发机构必须持续联机。

实际上，被分发的证书都有有限的生存期。对于因特网应用程序来说，到期时间通常为一年(Stein 1998)。这样的方法要求定期公布 CRL，但在检查证书时也要检查 CRL。实践证明，客户应用程序几乎没有查阅过 CRL，而只是假设证书在到期以前是有效的。看来实际涉及到因特网安全时，还有很多改善的余地。

9.4.2 安全组管理

许多安全系统都利用了特殊的服务机构，例如 KDC(密钥分发中心)或 CA(证书颁发机构)。这些服务机构的存在说明了分布式系统中的一个困难问题。首先，这些服务机构必须是可信的。要提高对安全服务机构的信任，就有必要对它们提供高度的保护以防止所有类型安全威胁。例如，CA 一旦受到损害，就不可能检验公钥的有效性，这会使整个安全系统完全失去价值。

另一方面，同样，许多安全服务机构也必须提供高可用性。例如，对于 KDC 来说，每当两个进程希望建立安全通道时，至少其中一个需要联系 KDC 以获得共享密钥。如果 KDC 不可用，那么安全通信就不能建立，除非可以用一种替代技术来建立密钥，例如 Diffie-Hellman 密钥交换技术。

解决高可用性的方法是复制。但另一方面，复制会使服务器更易受到安全攻击。我们已经讨论了安全组通信可以通过共享组成员之间的一个密钥的方式进行。实际上，没有单个的组成员能够破坏证书，这使组本身非常安全。仍需考虑的是实际管理一组复制服务器的方法。(Reiter 等 1994)提出了下面的解决方法。

需要解决的问题是确保进程要求加入组 G 时，该组的完整性没有遭到破坏。假设组 G 使用一个所有组成员共享的密钥 CKG 来加密组消息。此外，该组还使用一个公钥/私钥对(K_G^+,K_G^-)来与非组成员通信。

只要进程 P 希望加入组 G，它就发送一个标识 G 和 P 的加入请求 JR、P 的本地时间 T、一个生成的响应填充 RP 以及一个生成的密钥 $K_{P,G}$。RP 和 $K_{P,G}$ 使用该组的公钥 K_G^+ 共同加密，如图 9.35 中的消息 1 所示。RP 和 $K_{P,G}$ 的使用将在下面更详细地进行解释。加入请求 JR 由 P 签名，并与包含 P 的密钥的一个证书一起发送。我们已经使用了广泛应用的符号$[M]_A$来表示消息 M 已经由主体 A 签名。

一个组成员 Q 接收到这样的一个加入请求时，该成员首先对 P 进行身份认证，然后与

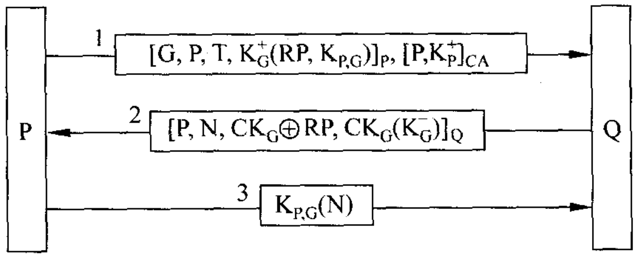


图 9.35 安全接纳一个新的组成员

另一个组成员进行通信来查看是否可以接纳 P 为组成员。P 的身份认证以通常方式依靠证书进行。时间戳 T 用来确定该证书在其发送时是否仍然有效(注意，我们还需要确认该时间没有受到篡改)。组成员 Q 检验该证书授权机构的签名，随后从该证书中提取出 P 的公钥以检查 JR 的有效性。到了这个时间点，就将遵循一个指定组的协议来查看是否所有的组成员都同意接纳 P 加入该组。

如果允许 P 加入该组，Q 就返回一个组许可进入消息 GA，如图 9.35 中的消息 2 所示，标识 P 并包含一个现时 N。响应填充 RP 用于加密该组的通信密钥 CK_G。此外，P 还需要使用 CK_G 加密的该组的私钥 K_G^-。消息 GA 随后由 Q 使用密钥 $K_{P,G}$ 签名。

进程 P 现在可以对 Q 进行身份认证，因为只有真正的组成员才能发现密钥 $K_{P,G}$。此协议中的现时 N 不用于安全，而是当 P 发回使用 $K_{P,G}$ 加密的 N(消息 3)时，Q 于是知道 P 已经接收到所有必需的密钥，因此现在确实加入到该组中了。

注意，如果不使用响应填充 RP、P 和 Q 也可以使用 P 的公钥加密 CK_G。然而，因为 RP 只能使用一次，也就是用于消息 GA 中组的通信密钥的加密，所以使用 RP 更为安全。如果 P 的私钥曾经泄露出来，那么就也可能泄露 CK_G，这样可能破坏所有组通信的保密性。

9.4.3 授权管理

分布式系统中的安全管理包括访问权限管理。到目前为止，我们几乎还没有直接涉及有关怎样把访问权限最初授予用户或用户组的方式，以及随后以不可伪造的方式维护这些权限的方式的问题。现在是弥补这一遗漏的时候了。

在非分布式系统中，访问权限管理相对容易。向该系统中添加一个新用户时，就向该用户赋予初始权限，例如，可以在指定的目录中创建文件和子目录、创建进程、使用 CPU 时间等。换句话说，用户的一个完整账号是为一台指定机器建立的，该机器中所有权限都已由系统管理员预先指定。

在分布式系统中，情况就很复杂，因为资源散布在多台机器上。如果要遵循非分布式系统的方法，就有必要为每个用户在每台机器上创建一个账号。实质上，这就是网络操作系统中所采用的方法。通过在中央服务器上创建唯一的账号可使情况稍微简单一些。每次用户访问某些资源或机器时都会查阅该服务器。

9.4.3.1 权能与属性证书

在分布式系统中广泛应用的一种好得多的方法是使用权能。正如我们上面简要解释过

的那样，权能是对于指定资源的一种不可伪造的数据结构，它确切指定它的拥有者关于该资源的访问权限。存在不同的权能实现方式。这里，我们简要讨论如 Amoeba 操作系统中所使用的实现方式(Tanenbaum 等 1986)。

Amoeba 是最初基于对象的分布式系统之一。其分布式对象的模型是远程对象的模型。换句话说，一个对象存在于一个服务器中，客户通过代理对该对象透明访问。为了调用对一个对象的操作，客户需将权能传送给其本地操作系统，然后该系统对该对象所在的服务器进行定位，随后对该服务器进行 RFC。

权能是一个 128 位的标识符，内部组织如图 9.36 所示。前 48 位由该对象的服务器在创建该对象时初始化，并有效形成了该对象的服务器的独立于机器的标识符，称为**服务器端口**(server port)。Amoeba 使用广播来定位服务器当前所在的机器。

48位	24位	8位	48位
服务器端口	Objec	tRights	Check

图 9.36 Amoeba 中的权能

其后的 24 位用于标识给定服务器上的对象。注意，服务器端口与对象标识符一起为 Amoeba 中的每个对象构成了一个 72 位的全系统范围的唯一标识符。下面的 8 位用于指定权能持有者的访问权限。最后，48 位的 check 字段用于使权能不可伪造，正如我们在下面几页中要解释的。

在创建一个对象时，其服务器选择一个随机 check 字段并将其同时存储在权能和其自身的表内。一个新权能的所有权限位最初都是 1，返回给客户的就是此所有者权能(owner capability)。权能在执行操作的请求中发送回服务器时，check 字段将被检验。

要创建一个有限权能，客户可以向服务器回传一个权能以及用于新权限的一个位掩码。服务器从其表中获得最初的 check 字段，将其与新的权限(必须是权能中权限的一个子集)进行 XOR 运算，然后通过一个单向函数运行该结果。

然后，服务器创建一个新的权能，object 字段中的值不变，但在 rights 字段具有新的权限位，在 check 字段中为单向函数的输出。新的权能随后被返回其调用者。如果愿意，客户可以将此新权能发送给另一个进程。

在图 9.37 中描述了生成有限权能的方法。在本例中，所有者关闭了除一个权限外的所有权限。例如，该有限权能可以允许读取对象，但不允许任何其他的操作。rights 字段的意义对每种对象类型是不同的，因为对象的合法操作本身也因对象类型的不同而不同。

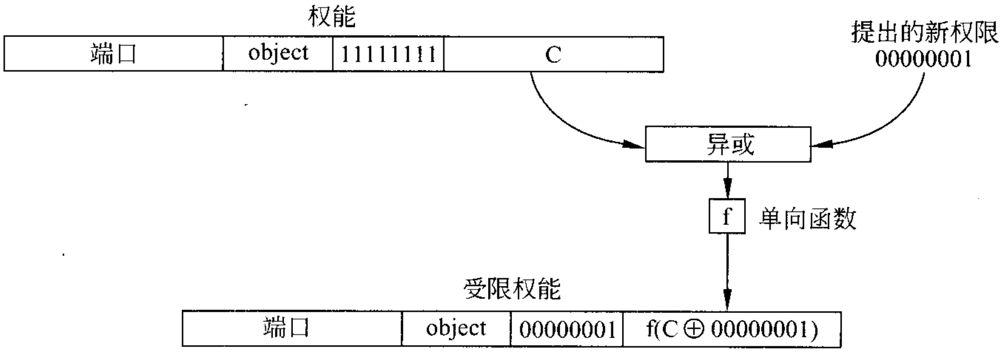


图 9.37 从一个所有者权能生成一个受限权能

有限权能返回服务器时,该服务器从 rights 字段看出其不是一个所有者权能,因为至少有一位被关闭了。然后该服务器从其表中取出最初的随机数,与该权能的 rights 字段进行 XOR 运算,通过单向函数运行该结果。如果该结果与 check 字段一致,则确认该权能是有效的。

从此算法中可以明显看出如果一个用户试图增加其没有的权限的用户只会使该权能无效。反向演算一个受限权能中的 check 字段来获取参数(图 9.37 中的 C XOR 00000001)是不可能的,因为函数 f 是单向函数。权能就是通过这种加密技术获得防止篡改的保护的。注意,在本质上,f 所做的事情与前面所讨论的计算消息摘要所做的事情相同。改变原始消息中的任何内容(例如反转一位)都会立即检测出来。

在现代分布式系统中使用的权能的一种概括是**属性证书**(attribute certificate)。与上面所讨论的用于检验公钥有效性的证书不同,属性证书用于列出应用于被标识实体的某些(属性,值)对。特别是,属性证书可以用于列出证书持有者关于该被标识资源的访问权限。

与其他证书相似,属性证书由称为**属性证书颁发机构**(attribute certification authority)的特殊证书颁发机构分发。与 Amoeba 的权能相比,这样的一个颁发机构与一个对象的服务器相对应。然而,一般来说,属性证书颁发机构和管理实体(为该实体已创建了一个证书)的服务器不需要是相同的。证书中列出的访问权限由该属性证书颁发机构签名。

9.4.3.2 委派

现在考虑下一个问题。一个用户希望打印一个他对其有只读访问权限的大文件。为了不过多打扰其他人,该用户向打印服务器发送一个请求,请求其在凌晨 2 点以后开始打印该文件。该用户没有将整个文件发送给打印机,而是将文件名发送给打印机,这样,在如有必要在实际需要时打印机可以将该文件复制到它的 spooling 目录上。

虽然此方法看起来不错,但存在一个主要问题:打印机一般对指定文件不具有适当的访问权限。换句话说,如果不采取特殊措施,一旦打印服务器希望读取该文件来进行打印,系统就会拒绝该服务器对该文件的访问。如果用户临时将其对该文件的访问权限委派给打印服务器,这一问题就可以解决。

对于实现计算机系统,特别是分布式系统中的保护来说,访问权限的委派是一项重要技术。其基本思想很简单:通过将某些访问权限从一个进程传送到另一个进程,使得在多个进程间分布工作变得更容易,而又不会对资源保护产生明显的影响。在分布式系统的情况下,进程可以在不同机器上,甚至不同管理域内运行,就像我们对 Globus 所进行的讨论一样。委派可以避免很多开销,因为保护通常可以在本地处理。

实现委派有多种方式。(Neuman 1993)中所描述的一种普通方法是利用代理。在计算机系统的安全的环境中,**代理**(proxy)是一个标记,允许其所有者使用与授予该标记的主体相同的或受限的权限和特权进行操作(注意这一代理概念与作为客户存根同义词的代理不同。虽然我们试图避免重叠使用术语,但这里是个例外,因为前面定义中的术语“代理”的使用过于广泛,不能忽略)。一个进程最多可以使用其所具有的权限和特权来创建一个代理。如果进程基于当前已有的一个代理创建一个新的代理,那么派生的代理至少具有与原始代理相同的限制,而且可能更多。

考虑委托的一般方法之前,先考虑下面两种方法。首先,如果 Alice 了解每个人,委托

就相对简单。如果她希望将权限委托给 Bob，那么她仅需构造一个证书说“Alice 说 Bob 具有权限 R”，例如$[A,B,R]_A$。如果 Bob 希望将这些权限中的一些传送给 Charlie，他会要求 Charlie 与 Alice 联系并向其请求一个适当的证书。

在第二种简单情况中，Alice 可以只构造一个证书说“此证书的持有者具有权限 R”。然而，在这种情况中，我们需要保护该证书以防违法的复制，正如在进程间安全传送权能中所做的那样。Neuman 的方法可以处理这种情况，而且还避免了 Alice 需要了解需要向其委派权能的每一个人。

Neuman 的方法中的代理有两部分，如图 9.38 所示。令 A 为创建该代理的进程。该代理的第一部分是一个集合 $C=\{R, S^+_{proxy}\}$，由一个已经由 A 委派的访问权限集合 R 以及一个用于对证书持有者进行身份认证的众所周知的密钥部分组成。下面我们将解释 S^+_{proxy}的使用。该证书带有 A 的签名 sig (A,C)，用于保护其免遭修改。第二部分包含该密钥的其他部分，用 S^-_{proxy}表示。将权限委派给另一个进程时保护 S^-_{proxy}防止暴露是最基本的。

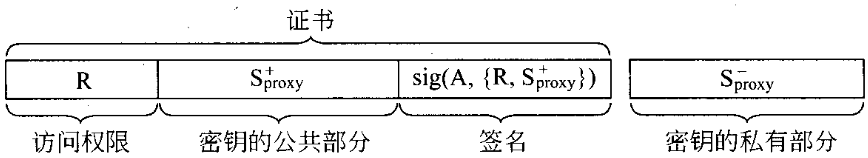


图 9.38 用于授权的常用代理结构

考虑代理的另一种方式如下。如果 Alice 希望将她的一些权限委派给 Bob，她就制作一个 Bob 可以使用的权限列表(R)。通过对该列表进行签名，她就可以防止 Bob 篡改该列表。然而，只有一个签名的权限列表通常是不够的。如果 Bob 希望使用其权限，他就必须证实其是否确实从 Alice 处获得该列表而不是从其他人处窃取的。因此，Alice 提出一个非常难以回答的问题(S^+_{proxy})，只有她知道(S^-_{proxy})的答案。当给出问题时，任何人都可以轻易地检验答案的正确性。在 Alice 添加其签名之前将该问题追加到权限表后。

当 Alice 委派她的一些权限时，会给予 Bob 签名的权限列表以及该难以回答的问题。她还在确保没有人能够窃听到的情况下给 Bob 该问题的答案。现在 Bob 拥有一个由 Alice 签名的权限列表，他可以在必要时将该列表移交给 Charlie。Charlie 会向其询问列表底部的那个难以回答的问题的答案。如果 Bob 知道该问题的答案，Charlie 就可以确信 Alice 确实已将该表列出的权限委派给 Bob 了。

此方案的一个重要特点是不需要向 Alice 请示。事实上，Bob 可以决定将列表上的(一些)权限传递给 Dave。要这样做的话，他还要告诉 Dave 该问题的答案，以使 Dave 能够证实该列表是由某个有资格的人移交给他的。Alice 根本不需要知道关于 Dave 的事。

用于委派和使用权限的一个协议如图 9.39 所示。假设 Alice 和 Bob 共享一个可用于加密他们相互发送的消息的密钥 $K_{A,B}$。于是，Alice 首先向 Bob 发送使用 sig(A,C)签名的证书 $C=\{R,S^+_{proxy}\}$(并重新以$[R,S^+_{proxy}]_A$表示)。没有必要加密该消息，它可以用明文发送。

只有该密钥的私用部分需要加密，如消息 1 中的 $K_{A,B}(S^-_{proxy})$所示。

现在假设 Bob 希望对指定服务器上的一个对象执行一个操作。还要假设 Alice 被授权可以执行该操作，并且她已经将那些权限委派给 Bob。因此，Bob 将其凭证以签名证书$[R,S^+_{proxy}]_A$的形式移交给该服务器。

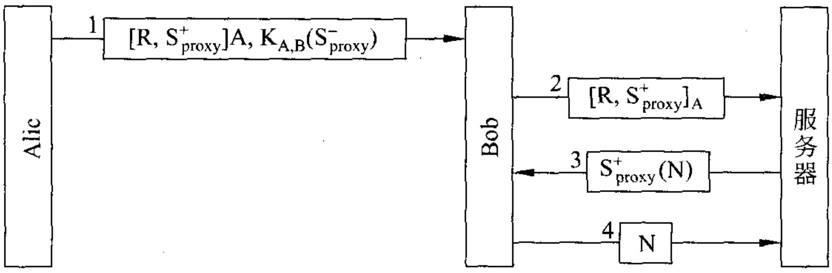


图 9.39　使用代理来授权和证实访问权限的所有权

这时，该服务器就能够检验 C 是否受到篡改，对权限表或该难以回答的问题的任何修改都会被注意到，因为两者共同由 Alice 签名。然而，该服务器还不知道 Bob 是否是该证书的合法所有者。要检验这一点，该服务器必须使用与 C 一起到来的密钥。

实现 S^+_{proxy} 和 S^-_{proxy} 有多种方式。例如，假设 S^+_{proxy} 是一个公钥，S^-_{proxy} 是相应的私钥。于是 Z 可以通过向 Bob 发送一个使用 S^+_{proxy} 加密的现时 N 来对其进行质询。通过解密 S^+_{proxy}(N) 并返回 N，Bob 证实了他知道该密钥并因此是该证书的合法持有者。实现安全委派还有其他方式，但基本思想始终是一样的：证明你知道一个秘密。

本章小结

安全在分布式系统中扮演一个非常重要的角色。分布式系统应该提供一些机制来实施多种不同的安全策略。开发和正确应用这些机制一般使安全成为了一项困难的工程实践。

三个重要问题可以区别开来。第一个问题是分布式系统应该提供建立进程间安全通信的功能。原则上，安全通道提供通信各方互相进行身份认证的方法，并保护消息在传输期间免受篡改。安全通道一般还提供机密性，使得除通信参与方之外，没有人能够读取该通道中通过的消息。

一个重要的设计问题是只使用对称加密系统（基于共享密钥），还是将其与公钥系统结合起来使用。当前的实践显示，为分配短期共享密钥进行的公钥加密的方法使用较多。短期共享密钥称为会话密钥。

安全分布式系统的第二个问题是访问控制和身份认证。身份认证以这样一种方式处理资源保护问题，即仅具有适当访问权限的进程能够实际访问和使用那些资源。访问控制总在进程通过身份认证后发生。

实现访问控制有两种方式。首先，每个资源能够保持一个访问控制表，正确列出每个用户或进程的访问权限；另一种方式是进程可以携带一个证书明确声明其对一个特定资源集合的访问权限。使用证书的主要好处是进程可以容易地将其票据传至另一个进程，也就是说，委派其访问权限。然而，证书具有通常难以吊销的缺点。

在移动代码的情况中处理访问控制时需要特别注意。除了要能够保护移动代码免受恶意主机危害之外，一般来说更重要的是保护主机免受恶意移动代码侵害。人们已经提出了若干个建议，其中沙箱是当前应用最广泛的一个。然而，沙箱有一定的限制性，而且基于真正的保护域的更灵活的方法也已设计出来。

安全分布式系统中的第三个问题关系到管理。本质上有两个重要的子专题：密钥管理和身份认证管理。密钥管理包括加密密钥的分配，可靠的第三方发布的证书对其扮演重要

的角色。关于身份认证管理的重要点在于属性证书和委派。

习 题

1. 分布式系统应该提供哪些机制作为对应用程序开发人员的安全服务，这些开发人员仅相信系统设计中的端到端参数吗(如第6章所述)?
2. 在 RISSC 方法中，是否所有安全性都集中于安全服务器上?
3. 假设要求您开发一个分布式应用程序，可使老师设立考试的。请给出至少三个可作为这种应用程序的安全策略中一部分的说明。
4. 将如图 9.12 中显示的身份认证协议中的消息 3 和消息 4 结合为 $K_{A,B}(R_B, R_A)$是否安全?
5. 在图 9.15 中，KDC 接收到对 Alice 和 Bob 可以共享的一个密钥的请求时，为什么不需要确切知道其在与 Alice 交谈?
6. 将现时(nonce)作为一个时间戳实现有什么错误?
7. 在 Needham-Schroeder 身份认证协议的消息 2 中，票据使用 Alice 和 KDC 之间共享的密钥进行加密。这一加密是必需的吗?
8. 我们能安全地修改如图 9.19 中所示的身份认证协议，使消息 3 仅由 R_B组成吗?
9. 设计一个使用公钥密码系统中的签名的简单身份认证协议。
10. 假设 Alice 希望向 Bob 发送一条消息 m。她没有使用 Bob 的公钥 K_B^+ 加密 m，而是生成了一个会话密钥 $K_{A,B}$，然后发送[$K_{A,B}(m)$，$K_B^+(K_{A,B})$]。为什么一般来讲，这种方法更好?(提示：考虑性能问题。)
11. 在图 9.23 的消息 6 中，时间戳的作用是什么，它为什么需要加密?
12. 请添加 Alice 与 Bob 之间的通信认证来完成图 9.23。
13. 如何表示访问控制矩阵中的角色转换?
14. 在 UNIX 文件系统中如何实现 ACL?
15. 一个公司如何加强 Web 代理网关的使用，并防止其用户直接访问外部 Web 服务器?
16. 参考图 9.31，将 Java 对象引用作为权能的使用，在多大程度上取决于 Java 语言?
17. 列举出需要本地资源接口的开发人员插入调用以启用和禁用特权时会遇到的三个问题，这些特权是用来防止通过移动程序进行未经授权的访问的。
18. 列举出使用集中式服务器进行密钥管理的一些优点和缺点。
19. Diffie-Hellman 密钥交换协议还可以用于建立一个三方之间的共享密钥。解释一下如何实现。
20. Diffie-Hellman 密钥交换协议中不存在身份认证。通过利用这一特性，一个恶意的第三方 Chuck 能够轻易地插入到 Alice 与 Bob 之间进行的密钥交换中，随后破坏安全性。解释这一行为是如何工作的。
21. 请简单说明在 Amoeba 中吊销权能的方式。
22. 限制会话密钥的生存期是否有意义?如果有，请举例说明其建立方式。
23. (**实验**)为由三台机器组成的分布式系统安全 Kerberos v5 环境，其中一台机器要运行 KDC。请确保可以在任意两台机器之间建立一个(Kerberos)telnet 连接，但只使用一个 KDC 注册口令。有关 Kerberos 的详细讨论见(German 2003)。

第10章 基于对象的分布式系统

从本章开始，我们将从对原理的讨论转向研究各种用于组织分布式系统的范型。第一个范型由分布式对象组成。在基于对象的分布式系统中，对象的概念在分布式实现中起着关键性的作用。从原理上讲，所有的一切都被作为对象处理，而客户将以调用对象的方式获得服务和资源。

分布式对象之所以成为重要的范型，是因为它相对比较容易把分布的特性隐藏在对象接口后面。此外，因为对象实际上可以是任何事物，所以它也是构建系统的强大范型。在本章中，我们将探讨如何把分布式系统的原理应用于许多著名的基于对象的系统。特别是，我们将介绍 CORBA、基于 Java 的系统和 Globe 的一些方面。

10.1 体系结构

面向对象构成了软件开发的重要范型。自从引入这个概念起，它就受到了普遍的欢迎。这是由于它自然就能够把软件构建到良好定义的、或多或少地独立的组件中。开发人员可以集中精力实现独立于其他开发人员的特定功能。

面向对象于 20 世纪 80 年代开始用于开发分布式系统。同样，在达到高度分布式透明性的同时通过远程服务器宿主独立对象的理念构成了开发新一代分布式系统的稳固基础。在本节中，我们首先将深入研究基于对象的分布式系统的通用体系结构，之后我们就可以看看如何在这些系统中部署特定的原理。

10.1.1 分布式对象

对象的关键特性在于它可以封装数据（称为**状态**(state)）以及对这些数据的操作（称为**方法**(method)）。可以通过**接口**(interface)来使用方法。一个进程要想访问或操纵对象的状态，除了通过对象的接口来调用可供它使用的方法之外别无他法，理解这一点很重要。一个对象可能实现多个接口。同样，给定一个接口定义，可能有多个对象为其提供实现。

把接口与实现这些接口的对象分隔开对于分布式系统是至关重要的。严格的隔离允许我们把接口放在一台机器上，而使对象本身驻留在另一台机器上。这种组织（如图 10.1 所示）通常称为**分布式对象**(distributed object)。

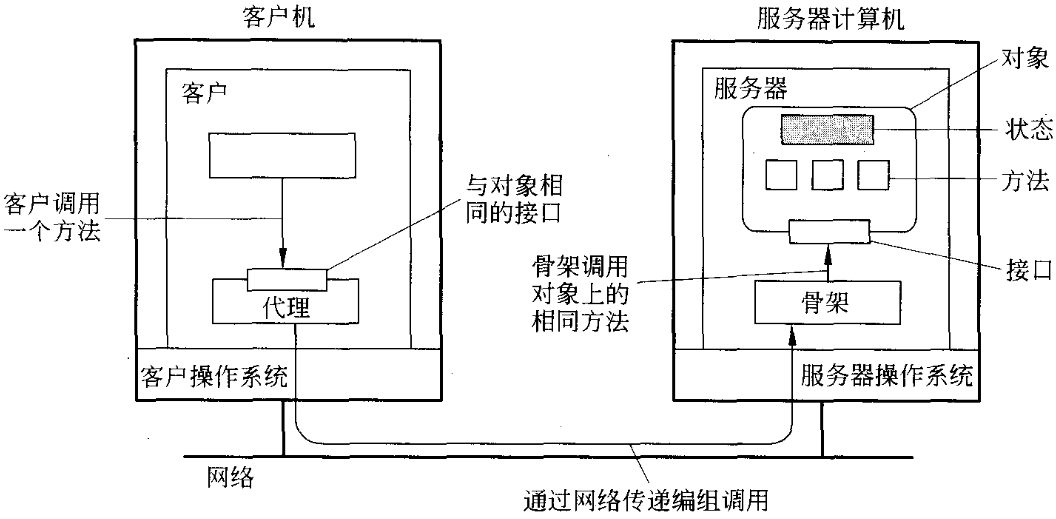


图 10.1　带有客户端代理的远程对象的常见组织

当客户**绑定**(bind)到一个分布式对象时，就会把这个对象的接口的实现(称为**代理**(proxy))加载进客户的地址空间中。代理类似于 RPC 系统中的**客户存根**(client stub)。它所做的唯一一件事是：把方法调用编组进消息中，以及对应答消息进行解组，把方法调用的结果返回给客户。实际的对象驻留在服务器计算机上，它在这里提供了与它在客户机上提供的相同接口。进入的调用请求首先被传递给服务器存根，服务器存根对它们进行解组，在服务器中的对象接口上进行方法调用。服务器存根还负责对应答进行编组，并把应答消息转发给客户端代理。

服务器端存根通常被称为**骨架**(skeleton)，因为它提供了明确的方式，允许服务器中间件访问用户定义的对象。实际上，它通常以特定于语言的类的形式包含不完整的代码，开发人员需要进一步对其进行特殊化。

大多数分布式对象的一个典型的但又有点违反直觉的特性是它们的状态不是分布式的：它驻留在单台机器上。在其他机器上只能使用被对象实现的接口。这样的对象也被称为**远程对象**(remote object)。在一般的分布式对象中，状态本身可能物理地分布在多台机器上，但是这种分布对于对象接口背后的客户是隐藏的。

10.1.1.1　编译时对象与运行时对象

分布式系统中的对象具有多种形式。最明显的形式是直接与语言级对象相关的形式，比如那些受 Java、C++ 或其他面向对象语言支持的对象，这些对象被称为编译时对象。在这种情况下，对象被定义为类的实例。**类**(class)依据具有数据元素以及该数据上的操作的模块来描述抽象类型(Meyer 1997)。

在分布式系统中使用编译时对象通常使得构建分布式应用程序要容易得多。例如，在 Java 中，可以完全通过类以及类实现的接口来定义一个对象。编译类定义将得到允许它实例化 Java 对象的代码。可以把接口编译进客户端和服务器端存根中，从而允许从远程机器调用 Java 对象。Java 开发人员在很大程度上可以不知道对象的分布情况：他只会看到 Java 编程代码。

编译时对象的一个明显的缺点是：它依赖于特定的编程语言。因此，构造分布式对象

的一种替代方式是：在运行时显式构造它。在许多基于对象的分布式系统中都沿用了这种方法，因为它独立于编写分布式对象所用的编程语言。特别是，可以从用多种语言编写的对象来构造应用程序。

在与运行时对象打交道时，对象的实际实现方式基本上悬而未决。例如，开发人员可能选择编写一个C库，其中包含许多函数，它们都可以处理公共的数据文件。其实质是：如何让这样一种实现看起来像是一个可以从远程机器调用其方法的对象。常用的方法是使用一个**对象适配器**(object adapter)，它充当实现的**包装器**(wrapper)，其唯一目的是使之看起来像是对象。"适配器"这个术语来源于(Gamma 等 1994)中描述的设计模式，它允许把接口转换成客户期望的某种事物。举一个对象适配器的例子，这个对象适配器动态绑定到上面提到的C库，并且打开一个代表对象当前状态的关联的数据文件。

对象适配器在基于对象的分布式系统中扮演着重要的角色。为了使包装尽可能简单，根据对象实现的接口单独定义它们。然后，可以把接口的实现限制到一个适配器，随后适配器可以使该接口可用于(远程)调用。适配器将负责调用请求被执行，从而把远程对象的映像提供给其客户。在本章后面将回过头来讨论对象服务器和适配器的组织。

10.1.1.2 持久对象与临时对象

除了语言级对象与运行时对象之分外，还有持久对象与临时对象之分。**持久对象**(persistent object)是指持续存在的对象，即使它目前没有包含在任何服务器进程的地址空间中也是如此。换句话说，持久对象不依赖于它当前的服务器。实际上，这意味着目前正在管理持久对象的服务器可以把对象的状态存储在辅助存储器中，然后退出。以后，新启动的服务器可以从存储器中把对象的状态读入它自己的地址空间中，并处理调用请求。与之相反，仅当宿主对象的服务器存在时，**临时对象**(transient object)才会存在。只要服务器退出，对象也会不复存在。对于是否需要持久对象过去存在很大的争议；一些人认为临时对象就足够了。为了把人们关注的焦点从中间件问题上移开，大多数基于对象的分布式系统同时提供了对这两类对象的支持。

10.1.2 示例：企业级Java Bean

Java编程语言以及关联的模型构成了众多分布式系统和应用程序的基础。它的普及可以归功于对面向对象的直接支持，并且结合了对远程方法调用的固有支持。我们将在本章后面讨论，Java提供了高度的访问透明性，这使得与(例如)结合使用C和远程过程调用相比，它更容易使用。

自从引入Java以来，就有一种强烈的动机，即提供一些工具以便于进行分布式应用程序开发。这些工具超越了语言支持，并且需要支持传统的多层客户-服务器体系结构的运行时环境。最终，在**企业级Java Bean**(Enterprise Java Beans，EJB)的开发中投入了大量的工作。

EJB实质上是由特殊服务器宿主的一种Java对象，该服务器为远程客户提供了多种不同的方式来调用那个对象。关键是这个服务器支持把应用程序功能与面向系统的功能隔开。后者包括查找对象、存储对象以及使对象成为事务的一部分等功能。在后面集中介绍

对象服务器时将讨论如何实现这种隔离。(Monson-Hafael 等 2004)详细描述了如何开发 EJB。在(Sun Microsystems 2005a)中可以找到相应的规范。

在牢记这种隔离的情况下,可以用图形表示 EJB,如图 10.2 所示。重要的问题是:EJB 嵌入在一个容器内,该容器实际上提供了对由应用程序服务器实现的底层服务的接口。容器可以或多或少地自动把 EJB 绑定到这些服务器上,这意味着正确的引用准备好供程序员使用。典型的服务包括那些用于远程方法调用(remote method invocation,RMI)、数据库访问(JDBC)、命名(JNDI)和消息传递(JMS)的服务。可以或多或少地自动利用这些服务,但是它们确实需要程序员区分 4 种 EJB:

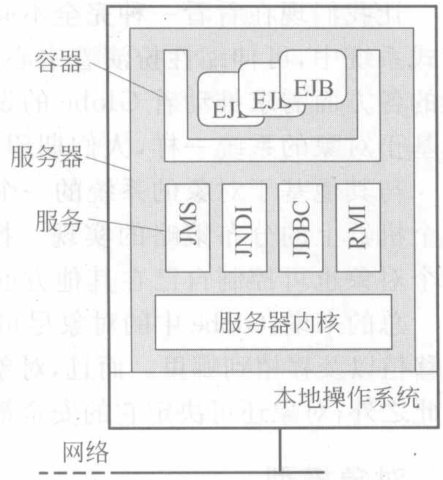


图 10.2 EJB 服务器的一般体系结构

(1) 无状态会话 bean;

(2) 有状态会话 bean;

(3) 实体 bean;

(4) 消息驱动的 bean。

顾名思义,**无状态会话 bean**(stateless session bean)是一种临时对象,一旦调用,它就会工作,之后它就会丢弃它执行提供给客户的服务所需的任何信息。例如,无状态会话 bean 可用于实现一种列出获得最高评级的图书的服务。在这种情况下,bean 通常包含一个提交给数据库的 SQL 查询。然后把结果转换成客户可以处理的特殊格式,这之后,就完成了它的工作并且会丢弃列出的图书。

与之相反,**有状态会话 bean**(stateful session bean)会维持与客户相关的状态。一个规范的示例是实现电子购物车的 bean,就像那些广泛部署用于电子商务的购物车一样。在这种情况下,客户通常能够把物品放入购物车内、删除项目,以及使用购物车转到电子收款处。这个 bean 反过来又通常会访问数据库,以获取当前价格以及关于仍然有现货的商品数量的信息。不过,其生存期仍是有限的,这就是为什么把它称为会话 bean 的原因:当客户完成时(可能已经调用了对象多次),将自动销毁 bean。

实体 bean(entity bean)可以被看作是长期存在的持久对象。因此,实体 bean 一般将存储在数据库中,同样,它们通常也是分布式事务的一部分。通常,实体 bean 存储下一次特定的客户访问服务器可能需要的信息。在用于电子商务的设置中,实体 bean 可用于记录顾客信息,例如:送货地址、开账单地址、信用卡信息等。在这些情况下,当一个客户登录时,将恢复与之关联的实体 bean,并将其用于进一步的处理。

最后,**消息驱动的 bean**(message-driven bean)用于编写那些应该对进入的消息做出反应(以及能够发送消息)的对象。消息驱动的 bean 不能直接被客户调用,但是适合于**发布-订购**(publish-subscribe)的通信方式,我们在第 4 章中简要讨论过它。简单地讲,当接收到特定的消息 m 时,服务器将自动调用消息驱动的 bean,这个服务器(而不是它宿主的应用程序)是以前订阅的。bean 包含用于处理消息的代码,这之后,服务器会简单地丢弃它。因此,消息驱动的 bean 被看作是无状态的。我们将在第 13 章中回过头详细讨论这类通信。

10.1.3 示例：Globe分布式共享对象

让我们现在看看一种完全不同的基于对象的分布式系统。在Globe这个基于对象的分布式系统中，可伸缩性扮演着中心角色。创建可支持海量用户和对象的大规模广域系统有关的各方面需求推动着Globe的设计。这个方法的基本出发点在于看待对象的方式。与其他基于对象的系统一样，人们期望Globe中的对象也会封装状态以及该状态上的操作。

与其他基于对象的系统的一个重要区别是：人们还期望对象封装那些规定对象状态在多个机器上的分布策略的实现。换句话说，每个对象决定它的状态如何分布在其副本上。每个对象也可控制自己在其他方面的策略。

总的来说，Globe中的对象尽可能多地承担职责。例如，对象决定它的状态如何移植、何时移植以及移植到哪里。而且，对象还决定它的状态是否会被复制，以及应该怎样进行复制。除此之外，对象还可决定它的安全策略和实现。下面的章节将描述如何实现这样的封装。

对象模型

与其他大多数基于对象的分布式系统不同，Globe没有采用远程对象模型。相反，Globe中的对象是可物理分布的，即对象的状态可在多个进程间分布和复制。这种组织结构如图10.3所示。图中显示了一个分布在4个进程上的对象，每一个进程又都分别运行在不同的机器上。Globe中的对象被称为**分布式共享对象**(distributed shared object)，以反映对象通常是在几个进程间共享的这一事实。Globe对象模型起源于Orca(Bal 1989)中使用的分布式对象。碎片对象(fragmented object)沿用了类似的方法(Makpangou等1994)。

绑定到分布式共享对象的进程可获得由那个对象提供的接口的本地实现。这样的本地实现称为**本地代表**(local representative)，或简称为**本地对象**(local object)。从原理上讲，无论是不是本地对象，都拥有对绑定的进程完全透明的状态。对象的所有实现细节都隐藏在提供给进程的接口后面。本地对象外部唯一可见的只有它的方法

Globe本地对象有两种类型。**基本本地对象**(primitive local object)不包含其他本地对象。与之相反，**组合本地对象**(composite local object)则包含多种(可能是组合的)本地对象。组合可用来构造实现分布式共享对象所需的本地对象。这个本地对象包括至少4个子对象，如图10.4所示。

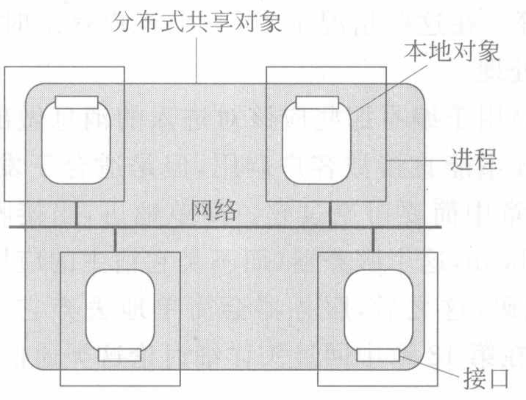


图10.3 Globe分布式共享对象的组织结构

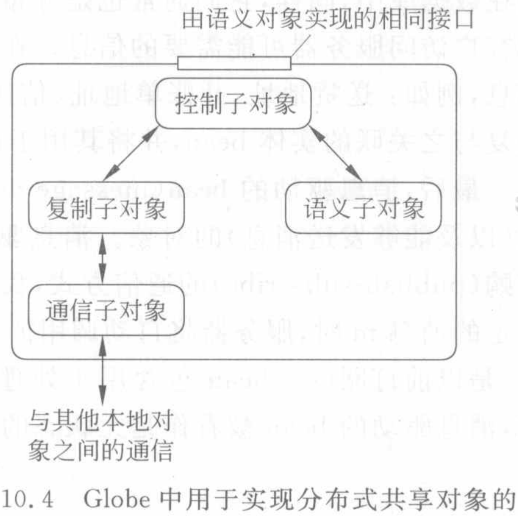


图10.4 Globe中用于实现分布式共享对象的本地对象的一般组织结构

语义子对象(semantics subobject)实现由分布式共享对象提供的功能。实质上,它对应于普通的远程对象,类似于 EJB。

通信子对象(communication subobject)用于向底层网络提供标准接口。这个子对象为面向连接或无连接的通信提供许多传递信元。也有更高级的通信子对象用于实现多播接口。通信子对象可以用于实现可靠的通信,而其他的只提供不可靠的通信。

事实上,对所有的分布式共享对象都至关重要的是**复制子对象**(replication subobject)。这个子对象实现对象的实际分布策略。就通信子对象而言,接口是标准化的。复制子对象负责准确决定由语义子对象提供的方法何时执行。例如,一个实现主动复制的复制子对象将保证所有的方法调用在每个副本中都以同样的顺序执行。这样,子对象将不得不与组成分布式共享对象的其他本地对象的复制子对象通信。

控制子对象(control subobject)是语义子对象的用户定义接口和复制子对象的标准化接口之间的媒介。除此之外,它还负责将语义子对象的接口输出到被绑定到分布式共享对象的进程。所以那个进程请求的方法调用都通过控制子对象编组并传递给复制子对象。

最终,复制子对象将允许控制子对象执行调用请求并将结果返回给进程。同样,远程进程的调用请求最后也会被传递给控制子对象。这样的请求随后被解组,然后由控制子对象执行调用,并将结果返回给复制子对象。

10.2 进　　程

对象服务器在基于对象的分布式系统中扮演着关键角色,也就是说,该服务器设计用于宿主分布式对象。在下面,我们首先重点介绍对象服务器的常规概念,然后我们将讨论开源 JBoss 服务器。

10.2.1 对象服务器

对象服务器是一种定制成支持分布式对象的服务器。常规的对象服务器与其他(更传统的)服务器之间的重要区别是:对象服务器本身不会提供特定的服务。特定的服务是由驻留在服务器中的对象实现的。实质上讲,服务器只提供了调用本地对象的方法,这是基于来自远程客户的请求进行的。因此,可以相对容易地更改服务,只需简单地添加和删除对象即可。

因此,对象服务器充当宿主对象的位置。一个对象由两部分组成:表示其状态的数据和用于执行其方法的代码。是否隔开这些部分或者方法实现是否由多个对象共享,都取决于对象服务器。此外,对象服务器调用其对象的方式也有区别。例如,在多线程服务器中,可能给每个对象分配单独的线程,或者把单独的线程用于每个调用请求。接下来将讨论这些及其他问题。

10.2.1.1 调用对象的替代方法

为了调用某个对象,对象服务器需要知道执行哪些代码,它应该操作哪些数据,它是否

应该启动单独的线程来负责调用,等等。一个简单的方法是假定所有对象看起来都是相似的,并且只有一种方式可用于调用对象。不幸的是,这样一个方法一般不灵活,并且通常会不必要地约束分布式对象的开发人员。

一种好得多的方法是使服务器支持不同的策略。例如,考虑临时对象。回忆可知:仅当其服务器存在时临时对象才存在,但是存在的时间可能更短一些。文件的内存中的只读副本通常可以被实现为临时对象。同样,计算器也可以被实现为临时对象。合理的策略是:在第一次调用请求时创建一个临时对象,一旦客户不再绑定到它就销毁它。

这种方法的优点是:仅当确实需要对象时,临时对象才需要服务器的资源。其缺点是:调用可能要花些时间才能完成,这是由于需要先创建对象。因此,有时一种替代策略是:在初始化服务器时创建所有的临时对象,其代价是会消耗一些资源,甚至在没有客户使用对象时也会如此。

服务器可以用一种类似的方式来遵循如下策略:把它的每个对象都放在它自己的内存段中。换句话说,对象既不共享代码,也不共享数据。当对象实现没有隔开代码和数据时或者当出于安全起见需要把对象隔开时,就可能需要这样一种策略。在后一种情况下,服务器将需要提供特殊的措施,或者需要提供广泛的支持,从底层操作系统到确保不违反分段界限。

一种替代方法是让对象至少共享它们的代码。例如,对于包含属于相同类的对象的数据库,可以通过只把类实现加载到服务器中一次来高效地实现它。当出现对对象调用的请求时,服务器只需从数据库中取出那个对象的状态,并且执行请求的方法即可。

同样,关于线程也有许多不同的策略。最简单的方法是只用一个控制线程来实现服务器。此外,服务器也可能具有多个线程,每个线程用于它的每个对象。无论何时出现对某个对象的调用请求,服务器都会把请求传递给负责那个对象的线程。如果线程目前很忙碌,暂时会对请求进行排队。

这种方法的优点是针对并发访问自动保护对象:通过与对象关联的单个线程来串行化所有的调用。多么简单!当然,也可能为每个调用请求使用单独的线程,这要求已经针对并发访问提供了对对象的保护。可以选择是根据需要创建线程还是让服务器维护一个线程池,这与为每个对象使用一个线程或者为每个方法使用一个线程无关。一般来讲,并没有一种最佳的策略。使用哪种策略依赖于线程是否可用、对性能会产生多大的影响,以及类似的因素。

10.2.1.2 对象适配器

关于如何调用一个对象的决策通常称为**激活策略**(activation policy),用以强调在许多情况下在可以实际地调用对象之前必须首先把对象自身引入服务器的地址空间中(也就是激活)。之后就需要一种机制,用于为每种策略组织对象。这样一种机制有时被称为**对象适配器**(object adapter),或者称为**对象包装器**(object wrapper)。最好把对象适配器看作是实现特定激活策略的软件。不过,主要问题是对象适配器是作为帮助分布式对象的开发人员的普通组件出现的,只需要为特定的策略配置它们。

对象适配器有一个或多个对象在其控制之下。由于服务器应该能够同时支持需要不同激活策略的对象,在同一个服务器中可能同时驻留多个对象适配器。当把调用请求传达给

服务器时，首先会把该请求分派给合适的对象适配器，如图10.5所示。

一个重要的观察是：对象适配器不知道它们控制的对象的特定接口。否则，它们永远不能是通用的。对于对象适配器来说，唯一重要的问题是它可以从调用请求中提取对象引用，随后把请求分派给引用的对象，但是现在要遵循特定的激活策略。同样如图10.5所示，它没有把请求直接传递给对象，而是由适配器把调用请求交给那个对象的服务器端存根。该存根(也称为骨架)通常是从对象的接口定义中产生的，它解组请求并调用合适的方法。

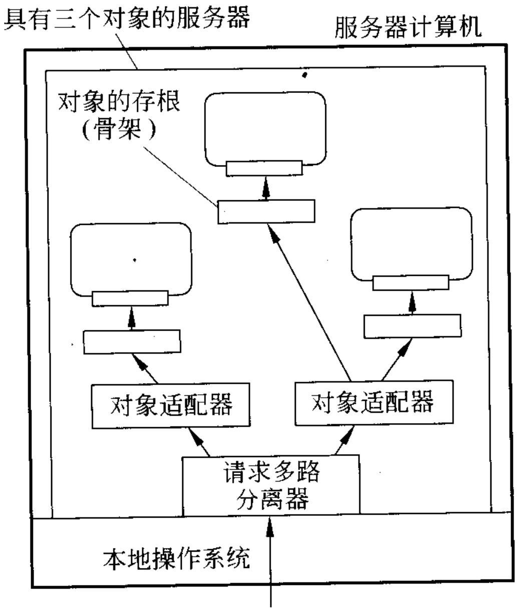


图10.5 支持不同激活策略的对象服务器的组织结构

可以通过在运行时简单地配置对象适配器，使之支持不同的激活策略。例如，在CORBA兼容的系统中(OMG 2004a)，可能指定当与对象关联的适配器停止以后该对象是否应该继续存在。同样，可以将适配器配置成生成对象标识符，或者让应用程序提供对象标识符。举最后一个例子，如我们以前所解释的，可以把适配器配置成工作在单线程或多线程模式下。

顺便说一下，注意：尽管在图10.5中我们谈到了对象，但是对这些对象实际上是什么我们一无所知。实际上，应该强调的是：作为这样一个对象的实现的一部分，服务器可能(间接地)访问数据库或者调用特殊的库例程。实现细节对只与骨架通信的对象适配器是隐藏的。因此，实际的实现可能与我们通常看到的语言级(即编译时)对象无关。鉴于此，通常采用一个不同的术语。用**伺服对象**(servant)这个通用术语来指示构成对象实现的代码段。从这一点上讲，只能将Java bean看作另一种伺服对象。

10.2.2 示例：Ice运行时系统

让我们看看在实际中如何处理分布式对象。我们简单地考虑Ice分布式对象系统，它已经被部分开发完成，这是为了响应商用基于对象的分布式系统的错综复杂性(Henning，2004)。在本节中，我们将重点介绍Ice对象服务器的核心，并把该系统的其他部分推迟到后面几节中介绍。

Ice中的对象服务器只是一个普通的进程，在初始化Ice**运行时系统**(runtime system，RTS)时就会简单地启动它。运行时环境的基础是由所谓的**通信器**(communicator)构成的。通信器是一个组件，用于管理许多基本资源，其中最重要的资源是由线程池构成的。同样，它将具有关联的动态分配的内存，等等。此外，通信器还提供了用于配置环境的工具。例如，它可能指定最大消息长度、最大调用重试次数，等等。

通常，对象服务器将只有一个通信器。不过，当需要把不同的应用程序完全隔离开并防止它们相互影响时，可以在同一个进程内创建单独的通信器(可能利用不同的配置)。至少，这样一种方法将隔开不同的线程池，使得如果一个应用程序消耗了它的所有线程，那么这不

会影响其他的应用程序。

通信器也可用于创建对象适配器,如图 10.6 所示。我们注意到该代码是简化的并且不完整。可以在(Henning 和 Spruiell 2005)中找到关于 Ice 的更多示例和详细信息。

```
main(int argc, char* argv[])  {
    Ice::Communicator          ic;
    Ice::ObjectAdapter         adapter;
    Ice::Object                objct;

    ic=Ice::initialize(argc, argv);
    adapter=
        ic->createObjectAdapterWithEnd Points("MyAdapter","tcp -p 1000");
    object=newMyObject;
    adapter->add(object, objectlD);
    adapter->activate();
    ic->waitForShutdown();
}
```

图 10.6 在 Ice 中创建对象服务器的示例

在这个示例中,我们首先创建和初始化运行时环境。当完成这个工作时,就会创建一个对象适配器。在这里,指示它在端口 10000 上侦听传入的 TCP 连接。注意:适配器是在刚才创建的通信器的环境中创建的。我们现在将创建一个对象,然后把该对象添加到适配器中。最后,**激活**(activate)适配器,这意味着在幕后激活一个线程,它将开始侦听传入的请求。

这段代码还没有显示激活策略中的大量区别。可以通过修改适配器的**属性**(property)来更改策略。一个属性族与维护一组适配器特有的线程相关,这些线程用于处理传入的请求。例如,人们可以指定应该总是只有一个线程,这实际上是把对添加到适配器中的对象的所有访问进行串行化。

此外,注意我们没有指定 MyObject。与以前一样,这可能是一个简单的C++ 对象,而且可能是一个访问数据库及其他外部服务(它们联合实现一个对象)的对象。通过用适配器注册 MyObject,对客户完全隐藏这样的实现细节,使得客户现在相信他们正在调用一个远程对象。

在上面的示例中,对象是作为应用程序的一部分创建的,然后将其添加到适配器中。这实际上意味着适配器可能需要同时支持许多对象,从而导致潜在的可伸缩性问题。一种替代解决方案是在需要对象时把它们动态加载到内存中。为此,Ice 提供了对称为**定位器**(locator)的特殊对象的支持。当适配器接收到对未显式添加的对象的传入请求时,将会调用定位器。在这种情况下,将把请求转发给定位器,它的工作是进一步处理请求。

为了使问题更具体,假设交给定位器一个对某个对象的请求,定位器知道该对象的状态存储在一个关系数据库系统中。当然,这里没有魔术:定位器被显式编程为处理这类请求。在这种情况下,对象的标识符可能对应于在其中存储状态的记录的键。定位器然后将简单地查找那个键,取出状态,然后能够进一步处理请求。

可以把多个定位器添加到适配器中。在这种情况下,适配器将记录哪些对象标识符属

于同一个定位器。使用多个定位器允许通过单个适配器来支持许多对象。当然，将需要在运行时加载对象（而不是它们的状态），但是这种动态行为可能使服务器本身相对比较简单。

10.3　通　信

现在把我们的注意力转向在基于对象的分布式系统中处理通信的方式上。不要奇怪，这些系统一般会给远程客户提供一些方法来调用对象。这种机制在很大程度上基于远程过程调用(remote procedure call，RPC)，我们在第4章中详细讨论过它。不过，在这可以发生之前，还有大量的问题需要处理。

10.3.1　把客户绑定到对象上

传统的RPC系统与支持分布式对象的系统之间的一个有趣的区别是：后者一般会提供系统级对象引用。可以在不同机器上的进程之间自由地传递这样的对象引用，例如，作为参数传递给方法调用。通过隐藏对象引用的实际实现（也就是说使之不透明），也许甚至是把它用作引用对象的唯一方式，与传统的RPC相比就增强了分布透明性。

当进程拥有对象引用时，在它可以调用所引用对象的任何方法之前，必须先绑定到该对象。绑定导致将代理存放到进程的地址空间中，并且实现一个包含进程可以调用的方法的接口。在许多情况下，绑定是自动完成的。当给底层系统提供一个对象引用时，它需要一种方法来定位管理实际对象的服务器，并把代理存放到客户的地址空间中。

利用**隐式绑定**(implicit binding)，将给客户提供一个简单的机制，允许它只使用对某个对象的引用来直接调用方法。例如，C++允许重要二元成员选择运算符（“－＞”），这允许我们像普通指针一样引入对象引用，如图10.7(a)所示。利用隐式绑定，将在把引用解析成实际对象的那一刻把客户透明地绑定到对象上。与之相反，利用**显式绑定**(explicit binding)，客户在可以实际地调用对象的方法之前，应该先调用一个特殊函数绑定到该对象上。显式绑定一般会返回一个指向代理的指针，然后该代理将在本地可用，如图10.7(b)所示。

```
Distr_object* obj_ref;          //Declare a systemwide object reference
obj_ref=...;                    //Initialize the refernce to a distrib. obj.
obj_ref→do_something();         //Implicitly bind and invoke a method
```
(a)

```
Distr_object obj_ref;           //Declare a systemwide object reference
Local_object* obj_ptr;          //Declare a pointer to local objects
obj_ref=...;                    //Initialize the reference to a distrib. obj.
obj_ptr=bind(obj_ref);          Explicitly bind and get ptr to local proxy
obj_ptr→do_comething();         Invoke a method on the lical proxy
```
(b)

图　10.7

(a) 只使用全局引用的隐式绑定的示例；(b) 使用全局和本地引用的显式绑定的示例

对象引用的实现

显然，对象引用必须包含足够的信息，以允许客户绑定到对象上。简单的对象引用将包括实际对象所驻留的机器的网络地址、标识管理对象的服务器的端点以及对象指示符。注意：这种信息的一部分将由对象适配器提供。不过，这种模式有许多缺点。

首先，如果服务器的计算机崩溃并且在恢复后把服务器分配给不同的端点，那么所有的对象引用都将变得无效。可以像在DCE中所做的那样解决这个问题：每台计算机上都有一个本地守护进程侦听已知的端点，并在端点表中记录服务器-端点分配。当把客户绑定到对象上时，我们首先询问守护进程服务器的当前端点是什么。这种方法要求我们把服务器ID编码进对象引用中，该对象引用可以用作进入端点表的索引。服务器反过来又总是要求用本地守护进程注册它自身。

不过，把服务器计算机的网络地址编码进对象引用中并不总是一个好主意。这种方法的问题是：如果没有验证指向服务器所管理对象的所有引用，那么永远都不能把该服务器移到另一台计算机上。一个明显的解决方案是：把用本地守护进程维护一个端点表的想法扩展到使用一个**定位服务器**（location server），它记录了对象的服务器目前正运行在哪些计算机上。然后，一个对象引用将包含定位服务器的网络地址，以及用于服务器的系统级标识符。注意：这种解决方案接近于我们在第5章中讨论过的无层次命名空间。

我们迄今为止不言而喻地假定：已经以某种方式把客户和服务器配置成使用相同的协议栈。这不仅意味着它们会使用相同的传输协议（例如，TCP）；而且意味着它们将使用相同的协议来编组和解组参数。它们还必须使用相同的协议来建立初始连接、处理错误和流量控制，等等。

如果我们在对象引用中添加了更多的信息，就可以安全地丢弃这种假定。这类信息可能包括：用于绑定到对象的协议的标识以及对象的服务器所支持的那些协议的标识。例如，单个服务器可能同时支持经由TCP连接传入的数据以及传入的UDP数据报。然后，客户负责为对象引用中标识的至少一个协议获取代理实现。

我们甚至可以扩展这种方法，并且在对象引用中包括一个**实现句柄**（implementation handle），它引用某个代理的完整实现，在绑定到对象时客户可以动态加载该代理。例如，实现句柄可以采用指向某个存档文件的URL形式，比如ftp://ftp.clientware.org/proxies/java/proxy-v1.1a.zip。然后，绑定协议只需要规定应该动态下载、解压缩、安装以及实例化这样一个文件。这种方法的好处是：客户无需担心它是否具有可用的特定协议的实现。此外，它允许对象开发人员自由地设计特定于对象的代理。不过，我们确实需要采取特殊的安全措施，以确保客户可以信任下载的代码。

10.3.2 静态远程方法调用与动态远程方法调用

在把客户绑定到对象之后，它就可以通过代理调用对象的方法。当涉及到编组和参数传递时，这样一个**远程方法调用**（remote method invocation）或者简称为**RMI**非常类似于RPC。RMI和RPC之间的本质区别是：如前所述，RMI一般支持系统级对象引用。此外，不必只使用通用的客户端和服务器端存根。相反，同样如我们所解释的那样，我们可以更轻松地纳入特定于对象的存根。

提供RMI支持的常用方式是在接口定义语言中指定对象的接口，这类似于RPC沿用的方法。此外，我们也可以利用基于对象的语言(如Java)，它将自动处理存根生成。这种使用预定义的接口定义的方法一般称为**静态调用**(static invocation)。在开发客户应用程序时，静态调用要求对象的接口是已知的。它还意味着如果接口发生变化，那么在客户应用程序可以利用新的接口之前必须重新编译它。

作为替代，也可以用更动态的方式来执行方法调用。在实际中，有时可以很方便地在运行时创建一个方法调用，也称为**动态调用**(dynamic invocation)。动态调用与静态调用之间的本质区别是：应用程序是在运行时选择它将调用远程对象上的哪个方法。动态调用一般具有如下形式：

```
invoke (object, method, input_parameters, output_parameters);
```

其中object标识分布式对象；method是一个参数，准确指定应该调用哪个方法；input_parameters是一种数据结构，用于保存该方法的输入参数的值；output_parameters也是一种数据结构，可用于存储输出值。

例如，考虑把整数int追加到文件对象fobject中，该对象为此提供了方法append。在这种情况下，静态调用将采用如下形式：

```
fobject.append(int)
```

而动态调用看起来将如下所示：

```
invoke(fobject, id(append), int)
```

其中操作id(append)为方法append返回一个标识符。

为了说明动态调用的有用性，考虑一个用于检查对象集的对象浏览器。假定浏览器支持远程对象调用。这样一个浏览器能够绑定到一个分布式对象上，然后把对象的接口展示给其用户。之后就可以要求用户选择一个方法并为其参数提供值，然后浏览器可以执行实际的调用。通常，应该开发这样一个对象浏览器以支持任何可能的接口。这样一个方法要求可以在运行时检查接口，并且可以动态地构造方法调用。

动态调用的另一个应用是一个批处理服务，可以把调用请求以及调用应该完成的时间传递给它。可以通过一个调用请求队列来实现该服务，并按完成调用的时间对该队列进行排序。服务的主循环将简单地等待，直到安排了下一个调用为止，从队列中删除请求，以及调用上面给出的invoke方法。

10.3.3 参数传递

由于大多数RMI系统都支持系统级对象引用，因此与RPC相比，它们在方法调用中传递参数一般不太受限制。不过，也有一些细微之处可能使得RMI比最初预想的更复杂，在下面几页中我们将简要讨论这方面的内容。

让我们首先考虑只有分布式对象的情况。换句话说，可以从远程计算机访问系统中的所有对象。在这种情况下，我们可以始终如一地在方法调用中把对象引用用作参数。引用按值传递，因此将会从一台机器复制到另一台机器。当方法调用的结果是给进程提供一个

对象引用时，以后可以根据需要把它简单地绑定到所引用的对象上。

不幸的是，只使用分布式对象可能导致效率极其低下，特别是当对象比较小时则更是如此，比如对象是整型或者（更糟糕的是）布尔型。客户执行的每个调用都不会作为对象共存于同一个服务器中，从而在不同地址空间之间或者（甚至更糟糕的是）在不同机器之间生成一个请求。因此，通常会用不同的方式处理指向远程对象的引用和那些指向本地对象的引用。

当把对象引用作为参数来调用一个方法时，仅当它引用一个远程对象时，才会复制并作为值参数传递它。在这种情况下，对象实际上是按引用传递的。不过，当这个引用指向一个本地对象时，它就是位于和客户相同地址空间中的对象，所引用的对象将作为一个整体被复制，并与调用一起进行传递。换句话说，对象是按值传递的。

图 10.8 中说明了这两种情况，该图显示客户程序运行在机器 A 上，服务器程序运行在机器 C 上。客户具有一个指向本地对象 O1 的引用，当调用机器 C 上的服务器程序时将把该对象用作一个参数。此外，客户还拥有一个指向远程对象 O2 的引用，该对象驻留在机器 B 上，它也会用作一个参数。当调用服务器时，将把 O1 的副本以及指向 O2 的引用的唯一一个副本一起传递给机器 C 上的服务器。

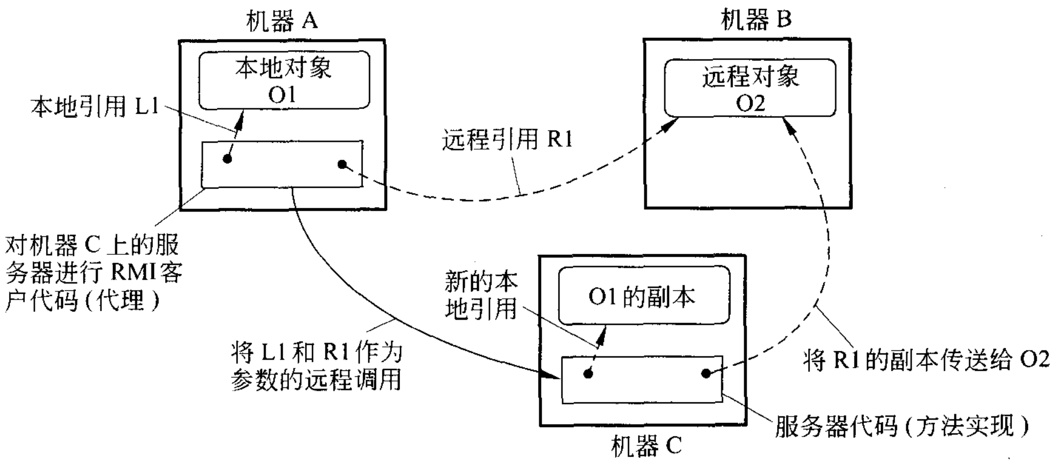


图 10.8 按引用或者按值传递对象时的情况

注意：我们是与指向本地对象的引用打交道还是与指向远程对象的引用打交道可能是高度透明的，比如在 Java 中就是这样。在 Java 中，仅当本地对象实质上具有与远程对象不同的数据类型时，才能看出它们之间的区别。否则，将会以非常相似的方式处理两种引用（Wollrath 等 1996）。另一方面，当使用传统的编程语言（如 C）时，指向本地对象的引用可以像指针一样简单，永远都不能用它来引用远程对象。

把对象引用作为参数来调用方法的副作用是：我们可能会复制对象。显然，隐藏这个方面是不可接受的，这使得我们将被迫在本地对象和分布式对象之间做出显式的区分。显示，这种区分不仅违反了分布透明性，而且使得更难以编写分布式应用程序。

10.3.4 示例：Java RMI

在 Java 中，已经将分布式对象集成进语言中。一个重要的目标是保留尽可能多的非分

布式对象的语义。换句话说,Java 语言开发人员的目标是获得高度的分布透明性。不过,如我们将看到的,Java 开发人员还决定使分布显而易见,因为高度透明性的实现效率极其低下,或者很难甚至不可能实现。

10.3.4.1 Java 分布式对象模型

Java 也采用远程对象作为分布式对象的唯一形式。回忆可知:远程对象是其状态总是驻留在单台机器上的分布式对象,但是其接口可供远程进程使用。接口是通过代理以常见的方式实现的,它提供了与远程对象完全相同的接口。代理本身以本地对象的形式出现在客户的地址空间中。

远程对象与本地对象之间只有几个区别,但是这些区别微妙且重要。首先,克隆本地或远程对象是不同的。克隆一个本地对象 O 将产生一个新对象,它具有与 O 相同的类型和完全相同的状态。因此,克隆会返回被克隆对象的准确副本。这些语义很难应用于远程对象。如果我们将要创建远程对象的一个准确副本,我们不仅必须克隆在其服务器上克隆实际的对象,而且必须克隆目前绑定到远程对象的每个客户上的代理。因此,克隆远程对象的操作只能由服务器执行。它会在服务器的地址空间中产生实际对象的准确副本。因此,不会克隆实际对象的代理。如果远程计算机上的客户希望访问服务器上的克隆对象,它必须首先再次绑定到那个对象。

10.3.4.2 Java 远程对象调用

很难在语言级看出本地对象与远程对象之间的区别,Java 在远程方法调用期间还可以隐藏大部分区别。例如,任何基本类型或对象类型都可以作为参数传递给 RMI,只要该类型可以编组即可。在 Java 术语中,这意味着它必须是**可串行化**(serializable)的。尽管在实际中大多数对象都可串行化,但是并非总是允许或者可能进行串行化。通常,与平台相关的对象(如文件描述符和套接字)不能进行串行化。

在 RMI 期间本地对象与远程对象之间的唯一区别是:本地对象是按值传递的(包括大对象,比如数组),而远程对象则是按引用传递的。换句话说,首先复制本地对象,然后把副本用作参数值。对于远程对象,将把指向对象的引用(而不是对象的副本)作为参数传递,图 10.8 中也说明了这一点。

在 Java RMI 中,指向远程对象的引用实质上是按我们在 10.3.3 节中所解释的那样实现的。这样一个引用包含服务器的网络地址和端点,以及服务器地址空间中的实际对象的本地标识符。这个本地标识符只由服务器使用。我们也解释过,指向远程对象的引用还需要编码客户和服务器用于通信的协议栈。为了理解在使用 Java RMI 时如何编码这样一个栈,认识到 Java 中的每个对象都是类的一个实例很重要。类反过来又包含一个或多个接口的实现。

实质上讲,远程对象是通过两个不同的类构建的。一个类包含服务器端代码的实现,我们称之为**服务器类**(server class)。这个类包含将在服务器上运行的远程对象的那一部分的实现。换句话说,它包含对象状态的描述,以及将操作该状态的方法的实现。服务器端存根(即骨架)是从对象的接口规范生成的。

另一个类包含客户端代码的实现,我们称之为**客户类**(client class)。这个类包含代理

的实现。像骨架一样，这个类也是从对象的接口规范生成的。最简单形式的代理所做的唯一一件事是：把每个方法调用转换成一条将发送给远程对象的服务器端实现的消息，并且在调用方法时把应答消息转换成结果。对于每个调用，它都会建立与服务器的连接，之后当调用完成时就断开它。出于此目的，如前所述，代理需要服务器的网络地址和端点。这种信息以及服务器上的对象的本地标识符总是被存储为代理状态的一部分。

之后，代理就会具有它所需的所有信息，允许客户调用远程对象的方法。在 Java 中，代理可以串行化。换句话说，有可能编组代理，并把它作为一系列字节发送给另一个进程，可以在其中解组它并用于调用远程对象上的方法。换句话说，代理可以用作指向远程对象的引用。

这种方法与 Java 集成本地和分布式对象的方式是一致的。回想一下，在 RMI 中，通过创建本地对象的一个副本来传递它，而远程对象则是通过系统级对象引用来传递的。代理只会被视作本地对象。因此，在 RMI 中有可能作为参数传递一个可串行化的代理。其副作用是：这样一个代理可以用作指向远程对象的引用。

在实际中，当编组一个代理时，它的完整实现（即它的所有状态和代码）将被转换成一系列字节。像这样编组代码不是非常高效，并且可能导致非常大的引用。因此，在 Java 中编组代理时，实际发生的是生成一个实现句柄，并且精确指定将需要哪些类来构造代理。其中一些类可能需要先从远程站点下载。实现句柄代替了作为远程对象引用一部分的编组代码。实际上，在 Java 中指向远程对象的引用一般只有几百个字节。

这种引用远程对象的方法非常灵活，并且是 Java RMI 的突出特性之一（Waldo 1998）。特别是，它允许特定于对象的解决方案。例如，考虑一个其状态偶尔才会发生变化的远程对象。我们可以通过在绑定时把整个状态复制到客户上，把这样一个对象转变成真正的分布式对象。每次客户调用一个方法时，它操作的都是本地副本。为了确保一致性，每次调用还会检查服务器上的状态是否发生了变化，如果是，就要刷新本地副本。此外，还将把修改状态的方法转发给服务器。远程对象的开发人员现在只需实现必要的客户端代码，并且在客户绑定到对象时动态地下载它。

仅当每个进程都在执行相同的 Java 虚拟机时，才能够作为参数传递代理。换句话说，每个进程都运行在相同的执行环境中。编组的代理会在接收端简单地进行解组，然后就可以执行它的代码。与之相反，例如，在 DCE 中，不可能传递存根，因为不同的进程可能运行在具有不同的语言、操作系统和硬件的执行环境中。作为替代，DCE 进程首先需要（动态地）链接一个本地可用的存根，该存根以前专门为进程的执行环境进行过编译。通过在 RPC 中把一个引用作为参数传递给存根，就有可能跨越进程界限来引用对象。

10.3.5 基于对象的消息传递

尽管 RMI 是基于对象的分布式系统中处理通信的首选方式，但是人们也发现消息传递是一种重要的替代方式。有多种基于对象的消息传递系统可用，并且如我们可能期望的那样，它们提供了完全相同的功能。在本节中，我们将深入研究 CORBA 消息传递，这部分是由于它还提供了一种组合方法调用和面向消息的通信的有趣方式。

CORBA 是一种著名的分布式系统规范。多年来，已经有多种实现在使用，尽管

CORBA本身将变得真正有多流行仍有待观察。不过，与普及性无关，CORBA规范非常全面(这对许多人来说也意味着它们非常复杂)。在认识到消息传递系统的普及性之后，CORBA迅速包括了一种消息传递服务的规范。

使得CORBA中的消息传递与其他系统不同的是其固有的基于对象的通信方法。特别是，消息传递服务的设计者需要保留通过调用对象来进行所有通信的模型。就消息传递来说，这种设计限制导致了两种形式的异步方法调用(此外，CORBA还提供了其他的形式)。

异步方法调用(asynchronous method invocation)类似于异步RPC：调用者会在启动调用之后继续工作，而不会等待一个结果。在CORBA的**回调模型**(callback model)中，客户提供了一个对象，它实现了一个包含回调方法的接口。底层通信系统可以调用这些方法，来传递异步调用的结果。一个重要的设计问题是：异步方法调用不会影响对象的原始实现。换句话说，把原始的同步调用转换成异步调用是客户的责任；展示给服务器的是通常的(同步)调用请求。

构造一个异步调用分两个步骤完成。首先，用两个新接口替换对象实现的原始接口，这两个新接口只能由客户端软件实现。其中一个接口包含客户可以调用的方法的规范。所有这些方法都不会返回一个值或者具有任何输出参数。第二个接口是回调接口。对于原始接口中的每个操作，它都包含一个将被客户的运行时系统调用的方法，用于在客户调用时传递关联方法的结果。

例如，考虑一个对象，它实现了只有一个方法的简单接口：

```
int add(in int i, in int j, out int k);
```

假定这个方法接受两个非负整数i和j，并且返回i+j作为输出参数k。假定：如果该操作没有成功完成，它就会返回−1。通过首先生成下面一对方法规范，把原始的(同步)方法调用转换成具有回调的异步方法调用(出于我们的目的，我们将选择方便的名称，而不是遵循OMG(2004a)中指定的严格规则)：

```
void sendcb_add(in int i, in int j);                //由客户向下调用
void replycb_add(in int ret_val, in int k);         //向上调用客户
```

实际上，将从客户调用的方法中删除原始方法规范中的所有输出参数，并且作为回调操作的输入参数返回它们。此外，如果原始方法指定了一个返回值，就会把这个值作为输入参数传递给回调操作。

第二步只编译已生成的接口。因此，将给客户提供一个存根，允许它异步地调用sendcb_add。不过，客户将需要提供回调接口的实现，在我们的示例中，它将包含方法replycb_add。最后一个方法是由客户的本地运行时系统(RTS)调用的，导致向上调用客户应用程序。注意：这些变化不会影响对象的服务器端实现。使用这个示例，在图10.9中总结了回调模型。

作为另一种选择，CORBA提供了**轮询模型**(polling model)。在这种模型中，客户可使用一系列操作轮询其本地RTS以得到传入的结果。和回调模型中一样，客户负责把原始的同步方法调用转换成异步调用。此外，通过从对象实现的原始接口中自动导出适当的方法规范，可以完成大多数工作。

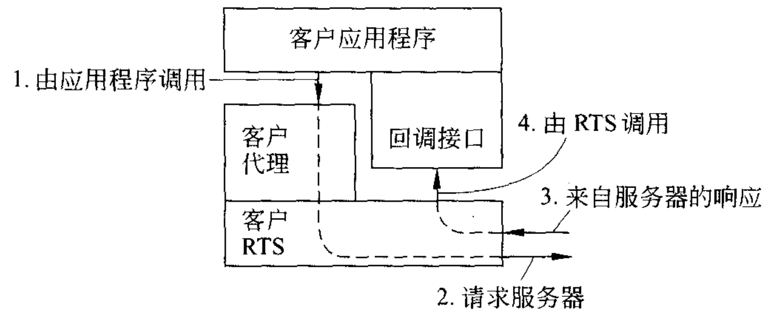


图 10.9　CORBA 用于异步方法调用的回调模型

回到我们的示例上来，方法 add 可得到如下两个生成的方法规范（再次方便地采用了我们自己的命名约定）：

```
void sendpoll add(in int i, in int j);                    //由客户调用
void replypoll_add(out int ret_val, out int k);           //也是由客户调用
```

轮询模型与回调模型之间最重要的区别是：replypoll_add 方法需要由客户的 RTS 实现。此实现可从接口规范中自动生成，就像我们为 RPC 所做的解释一样，客户端存根也是自动生成的。图 10.10 总结了轮询模型。同样，注意：对象的原始实现在服务器端不必更改。

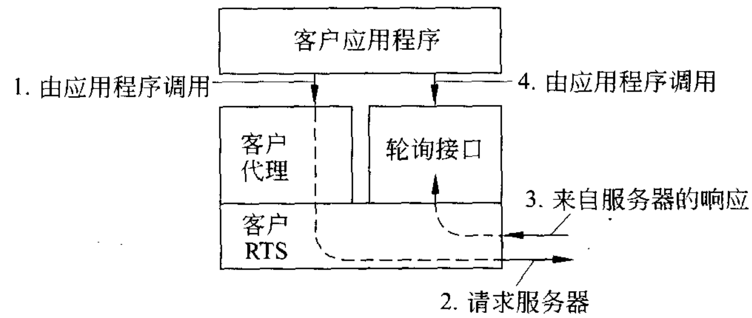


图 10.10　CORBA 用于异步方法调用的轮询模型

在迄今为止讨论过的模型中，还缺少一些内容。那就是客户与服务器间传送的消息，包括对异步调用的响应，在客户或服务器还没有运行的情况下都被底层系统存储了起来。幸运的是，与这些持久通信相关的绝大多数问题并不影响迄今为止已经讨论过的异步调用模型。需要的是建立一个消息服务器集合，允许临时存储消息（调用请求或响应），直到能够传输它们为止。

最后，CORBA 规范还包括用于所谓的路由器（router）的接口定义，它类似于我们在第 4 章中讨论过的消息路由器，例如，可以使用 IBM 的 WebSphere 队列管理器实现它。

此外，Java 还具有它自己的 **Java 消息传递服务**（Java Messaging service，JMS），它也与我们以前讨论过的非常相似（Sun Microsystems 2004a）。在第 13 章中，当我们讨论发布/订购范型时，将回过头来更详细地讨论消息传递。

10.4 命　名

在基于对象的分布式系统中,命名的有趣方面涉及支持对象引用的方式。我们已经在Java环境中描述了这些对象引用,在Java中,它们实际上对应于可移植的代理实现。不过,这是一个能够引用远程对象的语言相关的方式。再次以CORBA为例,让我们看看如何在一种语言中以与平台无关的方式提供基本的命名。我们还讨论了一种完全不同的模式,它用在Globe分布式系统中。

10.4.1 CORBA对象引用

CORBA的基础是其对象被引用的方法。当客户拥有对象引用时,它可以调用由所引用对象实现的方法。重要的是要区分客户进程用于调用方法的对象引用和由底层RTS实现的对象引用。

进程(可能是客户的或服务器的)只能使用一种与语言相关的对象引用的实现。在大多数情况下,这将采用指向对象的本地表示的指针形式。这种引用不能从进程A传递给进程B,因为它只在进程A的地址空间中有意义。相反,进程A可以先把指针编组成一个独立于进程的表示。这一操作由其RTS提供。一旦编组,就可以把引用传递给进程B并解组。注意,进程A和进程B可以是用不同语言编写的可执行程序。

相比之下,底层的RTS有它自己的独立于语言的对象引用的表示。这种表示甚至与它在进程间传递的用于交换引用的编组版本也不相同。重要的是当进程引用对象时,其底层RTS被隐式地传递了足够多的信息,从而知道实际上引用的是哪个对象。这种信息通常是由客户端和服务器端的存根传递的,这些存根从对象的接口规范中生成。

CORBA的早期版本中存在的问题之一是每个实现都能够决定它如何表示对象引用。因此,如前所述,如果进程A要向进程B传递一个引用,那么仅当两个进程使用相同的CORBA实现时,这一般才会成功。否则,进程A拥有的引用的编组版本对进程B使用的RTS毫无意义。

当前的CORBA系统都支持相同的独立于语言的对象引用表示法,称为**可互操作的对象引用**(Interoperable Object Reference)或**IOR**。CORBA实现是否在内部使用IOR并不重要。不过,当在两个不同的CORBA系统之间传递对象引用时,它是以IOR的形式传递的。IOR含有识别对象所需的全部信息。IOR的一般布局如图10.11所示,图中还给出了CORBA中使用的通信协议的特定信息。

每个IOR都以仓库标识符开头。此标识符被分配给一个接口,使得可以在接口仓库中存储和查找它。它用于在运行时检索接口信息,也可以帮助执行诸如类型检查或动态构造调用之类的操作。注意:如果此标识符是有用的,客户和服务器都必须能够访问相同的接口仓库,或者至少使用相同的标识符来确定接口。

每个IOR中最重要的部分都由所谓的**已标记配置文件**(tagged profile)组成。每个这样的配置文件都含有调用对象所需的完整信息。如果对象服务器支持多种协议,每种协议

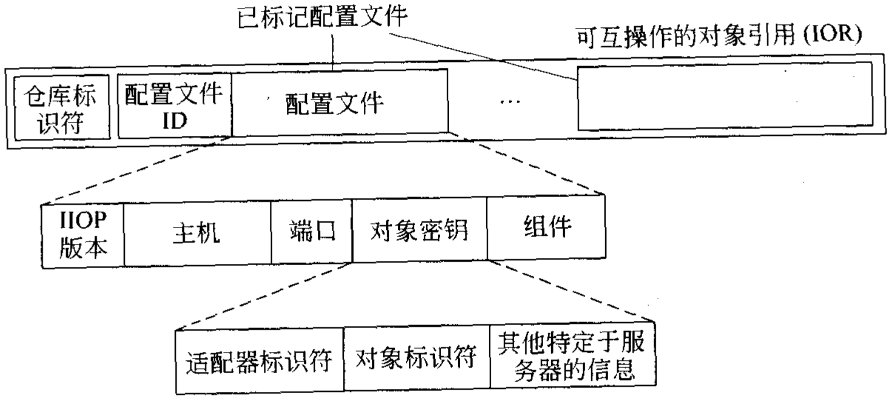


图 10.11 IOR 的组织以及用于 IIOP 的特定信息

的信息都可以记录在一个单独的已标记配置文件中。CORBA 使用**因特网 ORB 间协议**（Internet Inter-ORB Protocol，IIOP）来进行结点之间的通信（CORBA 为其基于对象的运行时系统使用**对象请求代理**（Object Request Broker，ORB）这个名称）。IIOP 实质上是用于支持远程方法调用的专用协议。图 10.11 中还给出了用于 IIOP 的配置文件的详细信息。

IIOP 配置信息用已标记配置文件中的配置文件 ID（ProfileID）字段确定。其主体包括5 个字段。IIOP 版本（IIOP version）字段确定此配置文件中使用的 IIOP 的版本。

主机（Host）字段是准确确定对象位于哪个主机上的字符串。主机可以用完整的 DNS 域名（例如 soling. cs. vu. nl）或者主机 IP 地址的字符串表示（例如 130. 37. 24. 11）来确定。

端口（Port）字段含有对象服务器侦听传入请求的端口号。

对象密钥（Object key）字段含有服务器相关信息，用于将传入的请求多路分离给相应的对象。例如，由 CORBA 对象适配器生成的对象标识符通常是该对象密钥的一部分。同时，这个密钥将确定特定的适配器。

最后，组件（Components）字段可以选择包含更多的信息，以便正确地调用所引用的对象。例如，这个字段可以包含指示应该如何处理引用的安全信息，或者指示在引用的服务器（临时）不可用的情况下应该怎么办。

10.4.2 Globe 对象引用

让我们现在看看一种引用对象的不同方式。在 Globe 中，每个分布式共享对象都会被分配一个全局唯一的对象标识符（OID），它是一个 256 位的串。Globe OID 是在第 5 章中定义的真正的标识符。换句话说，Globe OID 至多表示一个分布式共享对象，另一个对象永远都不会重用它，而每个对象也至多只有一个 OID。

Globe OID 只是用来比较对象引用的。例如，假设进程 A 和 B 都被绑定到一个分布式共享对象上。两个进程都可以请求它们被绑定到对象的 OID。当且仅当两个 OID 相同时，才认为 A 和 B 被绑定到了同一对象上。

与 CORBA 不同，Globe OID 不能用于直接联系一个对象。为了定位对象，需要在 Globe 定位服务中查找对象的联系地址。这种服务返回一个**联系地址**（contact address），

它对应于CORBA及其他分布式系统中使用的与地址相关的对象引用。尽管Globe使用它自己的特定地址服务，从原理上讲，第5章中讨论的任何地址服务也会执行这样的操作。

忽略一些次要细节，联系地址主要包含两个部分。第一部分是**地址标识符**(address identifier)，定位服务用它确定合适的叶结点，对关联的联系地址进行的插入和删除操作将被转发给该叶结点。记忆可知：由于联系地址是与位置相关的，从一个适当的叶结点开始插入和删除这些地址是很重要的。

第二部分由实际的地址信息组成，但此信息对定位服务是完全不透明的。对定位服务来说，地址只是一个字节数组，既可以表示实际网络地址，也可以表示编组的接口指针，甚至是完整的编组代理。

Globe目前支持两种类型的地址。**栈式地址**(stacked address)表示一个分层的协议簇，每层用一个三字段的记录表示，如图10.12所示。

字段	描述
协议标识符	一个表示(已知)协议的常数
协议地址	协议特有的地址
实现句柄	对类仓库中一个文件的引用

图10.12 栈式联系地址中的协议层的表示

协议标识符(Protocol identifier)是一个表示已知协议的常数。典型的协议标识符包括TCP、UDP和IP。**协议地址**(Protocol address)字段包含协议特有的地址，比如TCP端口号或者IPv4网络地址。最后，可以选择提供**实现句柄**(Implementation handle)，用于指定可以找到协议默认实现的位置。通常，实现句柄表示为一个URL。

第二种联系地址是**实例地址**(instance address)，它由两个字段组成，如图10.13所示。这种地址也含有**实现句柄**(implementation handle)，它仅仅是对能够找到本地对象的类仓库中一个文件的引用。当前绑定到对象的进程应加载此本地对象。

字段	描述
实现句柄	对类仓库中一个文件的引用
初始化串	用于初始化实现的串

图10.13 实例联系地址的表示

加载遵从与Java中的类加载相似的标准协议。在加载实现并创建本地对象之后，通过向对象传递**初始化串**(initialization string)进行初始化。此时，将完全解析对象标识符。

注意CORBA和Globe之间的对象引用中的区别，在基于对象的分布式系统中经常会见到这种区别。CORBA引用中包含在什么位置联系对象的准确信息，而Globe引用则需要执行额外的查找以检索该信息。这种区别也出现在像Ice这样的系统中，在CORBA中将其称为**直接**(direct)引用，而在Globe中则将其称为**间接**(indirect)引用(Henning和Spruiell，2005)。

10.5 同　　步

关于分布式系统中的同步只有几个问题，它们与处理分布式对象相关。在实际中，实现细节隐藏在接口后面的事实可能导致如下问题：当一个进程调用一个（远程）对象时，它不知道该调用是否会导致调用其他对象。因此，如果一个对象阻止并发访问，我们可能就会具有调用进程所不知道的一组级联锁，如图 10.14(a)所示。

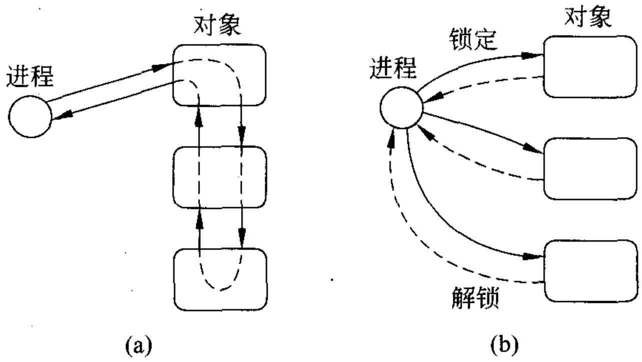


图 10.14　用于锁定对象的控制流程中的区别

与之相反，当处理诸如受锁保护的文件或数据库表之类的数据资源时，控制流程的模式实际上对于使用这些资源的进程是可见的，如图 10.14(b)所示。因此，当出现问题时，进程也可以在运行时施加更多的控制，比如当它认为发生死锁时就放弃锁。注意，事务处理系统一般沿用图 10.14(b)中所示的模式。

因此，在基于对象的分布式系统中，知道同步在哪里以及何时发生是重要的。同步的一个明显位置是在对象服务器中。如果对同一个对象的多个调用请求到达，服务器可以决定串行化这些请求(并且当它自身需要进行远程调用时可能在对象上保持一个锁)。

不过，在调用客户崩溃的情况下，让对象服务器维持锁将会使事情复杂化。鉴于此，也可以在客户端执行锁定，Java 中已经采用了这种方法。不幸的是，这种模式具有它自己的缺点。

如前所述，在 Java 中，本地对象与远程对象之间的区别通常难以界定。当通过把对象的方法声明为 synchronized 来保护对象时，事情将变得更复杂。如果两个进程同时调用一个同步方法，只有其中一个进程将执行其工作，而另一个进程将被阻塞。这样，我们可以确保对对象的内部数据的访问是完全串行化的。也可以在对象内部阻塞进程，并等待某个条件变为真。

从逻辑上讲，远程对象中的阻塞很简单。假设客户 A 调用远程对象的一个同步方法。为了使得访问远程对象看起来总是与访问本地对象完全一样，有必要的实现对象接口的以及 A 可以直接访问的客户端存根中阻塞 A。同样，在可以把不同机器上的另一个客户的请求发送给服务器之前，也需要在本地阻塞它。其结果是：我们需要同步不同机器上的不同客户。如第 6 章所讨论的，分布式同步可能相当复杂。

一种替代方法是：只允许在服务器上进行阻塞。从原理上讲，这种方法会工作得很好，

但是当服务器在处理客户的调用时，如果客户崩溃就会发生问题。如第 8 章中所讨论的，我们可能需要相对先进的协议来处理这种情况，它可能会影响影响远程方法调用的总体性能。

因此，Java RMI 的设计者选择在远程对象上仅限于阻塞代理（Wollrath 等 1996）。这意味着相同进程中的线程将被阻止并发访问相同的远程对象，但是不会阻止不同进程中的线程。显然，这些同步语义不易掌握：在语义级（也就是说在阅读源代码时）我们可能会看到一种美妙、清晰的设计。仅当实际地执行分布式应用程序时，才可能观察到应该在设计时处理的非预期行为。我们在这里看到了一个清楚的示例，为分布透明性奋斗并不是我们要走的理想道路。

10.6　一致性与复制

许多基于对象的分布式系统沿用复制对象的传统方法，实际上是把它们视作数据及其特殊操作的容器。因此，当我们考虑在支持 Java bean 的系统或者遵从 CORBA 的分布式系统中如何处理复制时，除了我们在第 7 章中讨论过的内容之外，实际上并没有太多新内容需要学习。

出于这个原因，我们将重点关注几个关于一致性和复制的特定主题，与其他系统中相比，它们在基于对象的分布式系统中具有更加深远的意义。我们首先将考虑一致性，然后转向介绍复制调用。

10.6.1　入口一致性

如第 7 章中所提到的，分布式对象的以数据为中心的一致性自然具有入口一致性形式。回想一下，在这种情况下，目标是使用同步变量（例如，以锁的形式）对共享数据上的操作进行分组。当对象自然地组合数据以及数据上的操作时，在调用期间锁定对象将串行化访问并保持它们一致。

尽管从概念上讲将锁与对象关联起来很简单，但是在复制对象时它不一定会提供正确的解决方案。在实现入口一致性时需要解决两个问题。第一个问题是：我们需要一种方式来阻止同一个对象上的多个调用并发执行。换句话说，在执行对象上的任何方法时，不可能执行其他任何方法。这种要求确保的确会串行化对对象的内部数据的访问。简单地使用本地锁定机制即可确保这种串行化。

第二个问题是：就复制对象而言，我们需要确保对对象的复制状态的所有改变都是相同的。换句话说，我们需要确保在相同时间不会在不同副本上发生两个独立的方法调用。这种要求意味着我们需要对调用进行排序，使得每个副本都会看到所有的调用具有相同的顺序。这种要求一般可以用以下两种方式之一来满足：(1)使用基于主版本的方法；(2)使用对副本的完全有序的多播。

在许多情况下，设计复制对象的方法如下：首先设计单个对象，可能通过本地锁定阻止对它进行并发访问，然后复制它。如果我们使用基于主版本的模式，那么应用程序开发人员需要做的额外工作就是串行化对象调用。因此，通常可以方便地假定底层中间件支持完全

有序的多播，因为这不需要客户做出任何改变，它也不需要应用程序开发人员做额外的编程工作。当然，中间件如何实现完全有序的多播应该是透明的。为了让所有的应用程序可能知道这一点，其实现可能使用基于主版本的模式，但是基于 Lamport 时钟同样很好。

不过，即使底层的中间件提供了完全有序的多播，也可能需要做更多的工作来保证有序的对象调用。问题出在粒度性之一上：尽管对象服务器的所有副本都可能以相同的顺序接收调用请求，我们还需要确保这些服务器中的所有线程也会以正确的顺序处理这些请求。图 10.15 描述了这个问题。

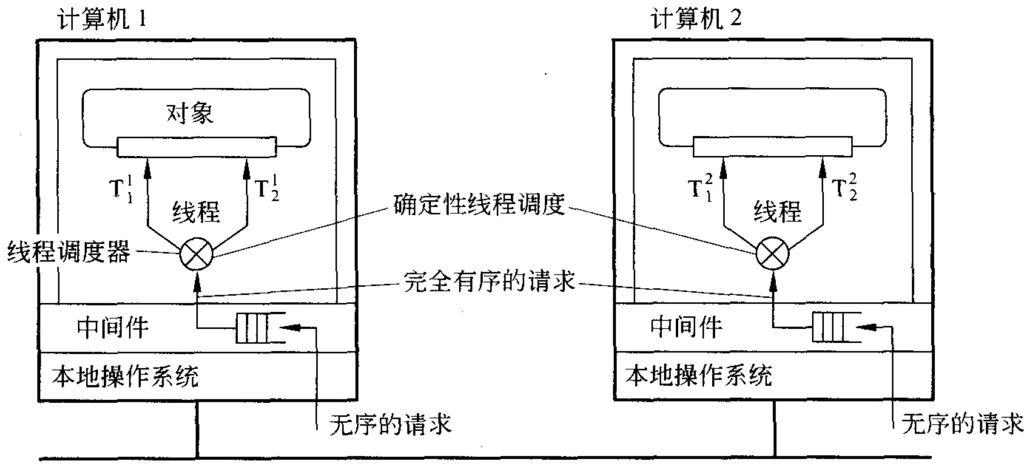


图 10.15 用于复制的对象服务器的确定性线程调度

多线程(对象)服务器只是简单地获取传入的请求，把它传递给可用的线程，并等待下一个请求到来。服务器的线程调度器随后把 CPU 分配给可运行的线程。当然，如果中间件已经尽量提供了完全有序的请求传送，线程调度器就应该以确定性方式工作，以便不会混淆同一个对象上的方法调用的顺序。换句话说，如果图 10.15 中的线程 T_1^1 和 T_1^2 处理相同的传入(复制的)调用请求，就应该在 T_1^1 和 T_1^2 之前分别对它们进行调度。

当然，并不需要以确定性方式简单调度所有的线程。从原理上讲，如果我们已经具有完全有序的请求传送，就只需要确保以与传送相同的顺序来处理对同一个复制对象的所有请求。这样一个方法将允许并发处理对不同对象的调用，而不会受到线程调度器的进一步限制。不幸的是，只有几个系统支持这种并发性。

(Basile 等 2002)中描述的一种方法可以确保在每个副本上以相同的顺序调用共享相同(本地)锁的线程。在基于主版本模式的基础上，复制服务器可以确定对于特定的锁应该优先执行哪个线程。(Basile 等 2003)中描述了一种改进方法，它可以避免服务器之间的频繁通信。注意，因此，不共享锁的线程可以在每个服务器上并发工作。

这种模式的一个缺点是它工作在底层操作系统级别上，这意味着每个锁都需要管理。通过提供应用程序级信息，只标识出那些需要串行化复制对象访问的锁，可以获得巨大的性能改进(Taiani 等 2005)。当我们讨论 Java 的容错性时将回过头来讨论这些问题。

复制框架

大多数基于对象的系统的一个有趣方面是：对象技术有一个天性，它通常可以清晰地把设计功能与处理额外的功能问题(如复制)隔离开。如我们在第 2 章中所解释的，截取器构成了完成这种隔离的强大机制。

(Babaoglu 等 2004)描述了一个框架,他们在该框架中使用截取器复制用于 J2EE 服务器的 Java bean。他们的想法相对比较简单：在三个不同的时间点截获对对象的调用,如图 10.16 所示：

(1) 在客户端,刚刚要把调用传递给存根之前。

(2) 在客户的存根内,其中截获构成了复制算法的一部分。

(3) 在服务器端,刚刚要调用对象之前。

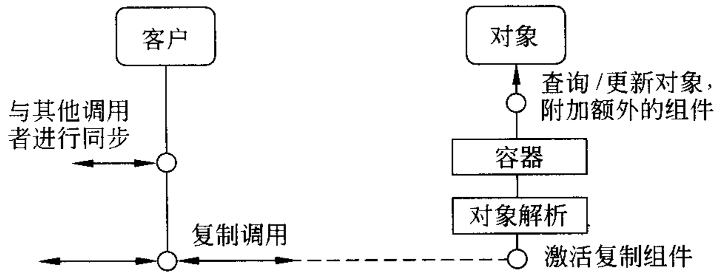


图 10.16　在 EJB 环境中把复制算法与对象隔离开的一般框架

当证实调用者被复制时,就需要进行第一次截获。在这种情况下,可能需要与其他调用者进行同步,因为我们可能像以前讨论过的那样处理复制的调用。

一旦决定可以执行调用,客户端存根中的截取器就可以决定把请求转发到哪里,或者可能实现一种故障转移机制,以用于副本不可访问时。

最后,服务器端截取器处理调用。事实上,这个截取器被一分为二。在第一个时间点,就在请求到达之后和把它交给适配器之前,复制算法获得控制。然后它可以分析打算把请求发送给谁,以允许它激活。如果必要,它需要的任何复制对象都将执行复制。第二个时间点出现在调用刚好发生之前,例如,允许复制算法获取和设置复制对象的属性值。

一个有趣的方面是：可以独立于任何复制算法建立这个框架,从而导致把对象功能与对象的复制完全隔离开。

10.6.2　复制的调用

需要解决的另一个问题是复制调用的问题。考虑一个对象 A 调用另一个对象 B,如图 10.17 所示。假定对象 B 还调用另一个对象 C。如果 B 被复制,从原理上讲,B 的每个副本都会独立地调用 C。问题是 C 现在被调用了多次,而不只是一次。如果 C 上的调用方法导致 100 000 美元的转账,那么显然,迟早会有人抱怨。

并没有许多通用的解决方案可用于解决复制调用的问题。一个解决方案是：简单地禁止它(Maassen 等 2001),当性能利害攸关时这样做是有意义的。不过,当为容错性考虑而进行复制时,可以部署(Mazouni 等 1995)提议的下面这个解决方案。他们的解决方案独立于复制策略,即如何使副本保持一致的准确详细信息。其实质是在执行(复制)对象的上面提供一个知道复制的通信层。当复制对象 B 调用另一个复制对象 C 时,首先通过 B 的副本给调用请求分配相同的唯一标识符。此时,B 的副本的协调器将把它的请求转发给对象 C 的所有副本,而 B 的其他副本将会阻止它们的调用请求的副本,如图 10.18(a)所示。结果是只会有一个请求被转发给 C 的每个副本。

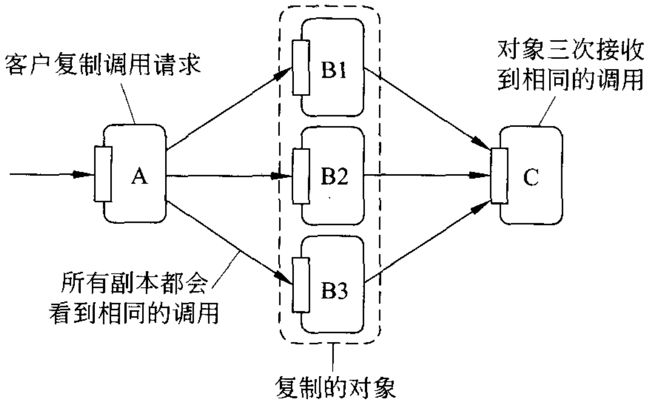


图 10.17　复制方法调用的问题

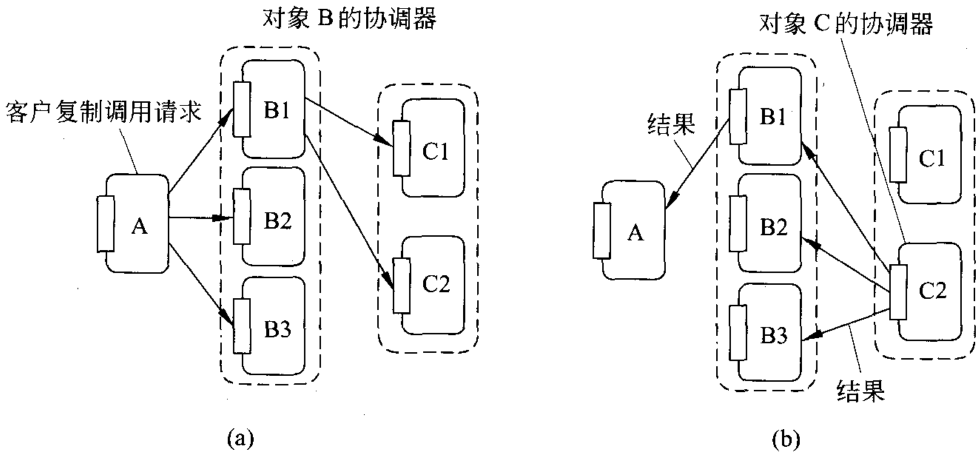


图　10.18

(a) 把调用请求从一个复制对象转发给另一个复制对象；(b) 把应答从一个复制对象返回给另一个复制对象

使用相同的机制来确保只有一条应答消息被返回给 B 的副本。这种情况如图 10.18(b)所示。C 的副本的协调器注意到它正在处理由 C 的每个副本生成的复制应答消息。不过，只有协调器会把该应答转发给对象 B 的副本，而 C 的其他副本会阻止它们的应答消息的副本。

当 B 的副本接收到它转发给 C 或者由于它不是协调器而阻止的调用请求的应答消息时，之后就会把应答交给实际的对象。

实质上讲，刚才描述的模式基于使用多播通信，但它会阻止不同的副本多播相同的消息。因此，它实质上是一种基于发送方的模式。一种替代解决方案是：让接收副本检测属于同一个调用的传入消息的多个副本，并且只把一个副本传递给其关联的对象。我们把这种模式的一些具体细节留作习题。

10.7 容 错 性

与复制一样，大多数基于对象的分布式系统中的容错性都使用与其他分布式系统中相同的机制，它们遵循的是我们在第 8 章中讨论的原理。不过，当谈到标准化时，CORBA 被

证明提供了最全面的规范。

10.7.1 示例：CORBA 的容错性

CORBA 中处理失败的基本方法是把对象复制到**对象组**(object group)中。这种组由相同对象的一个或多个完全相同的副本组成。不过，对象组可以当做单个对象来引用。这个组提供与其包含的副本相同的接口。换句话说，复制对客户是透明的。不同的复制策略都受到支持，包括主-备份复制、主动复制和基于法定数量的复制。这些策略都在第 6 章中讨论过。对象组还有各种其他属性，其细节可参见(OMG 2004a)。

为了尽可能多地提供复制与失败处理的透明性，对象组不应该与普通 CORBA 对象有区别，除非应用程序有特殊要求。在这方面，一个重要的问题是如何引用对象组。下面的步骤是使用一种特殊的 IOR，称为**可互操作的对象组引用**(Interoperable Object Group Reference, IOGR)。IOGR 与普通 IOR 的关键区别是：IOGR 含有对不同对象的多重引用，在同一个对象组中引人注目地进行复制。相比之下，IOR 也可能含有多重引用，但是都是引用同一个对象，尽管有可能使用不同的访问协议。

每当客户把 IOGR 传递给其运行时系统(RTS)时，该 RTS 就试图绑定到所引用的副本之一上。在使用 IIOP 的情况下，RTS 很有可能使用在 IOGR 的 IIOP 配置文件中找到的附加信息。就像我们前面讨论过的那样，此类信息存储在**组件**(Components)字段中。例如，一个特定的 IIOP 配置文件可能会引用对象组的主版本或备份，如图 10.19 所示。方法是分别使用单独的 TAG_PRIMARY 和 TAG_BACKUP 标签。

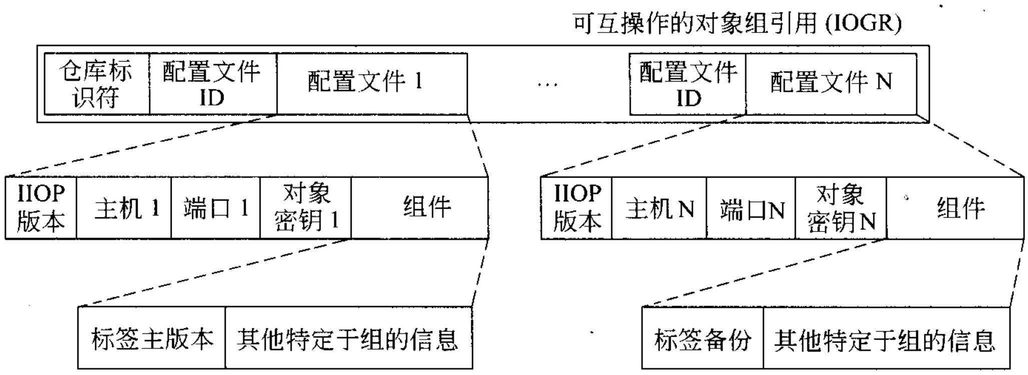


图 10.19 拥有主版本和备份的对象组的 IOGR 的可能结构

如果绑定到副本之一上失败，客户 RTS 可能继续尝试绑定到其他的副本上，根据某种策略选择它最适合的下一个副本。对客户来说，绑定过程是完全透明的；它看上去就好像客户绑定到一个普通 CORBA 对象上那样。

示例体系结构

为了支持对象组并且处理更多的失败管理，需要向 CORBA 中添加组件。CORBA 容错版本的一个可能的体系结构如图 10.20 所示。这个体系结构是从 Eternal 系统派生而来的(Moser 等 1998；Narasimhan 等 2000)。Eternal 系统在 Totem 可靠组通信系统的基础上提供了容错性基础结构(Moser 等 1996)。

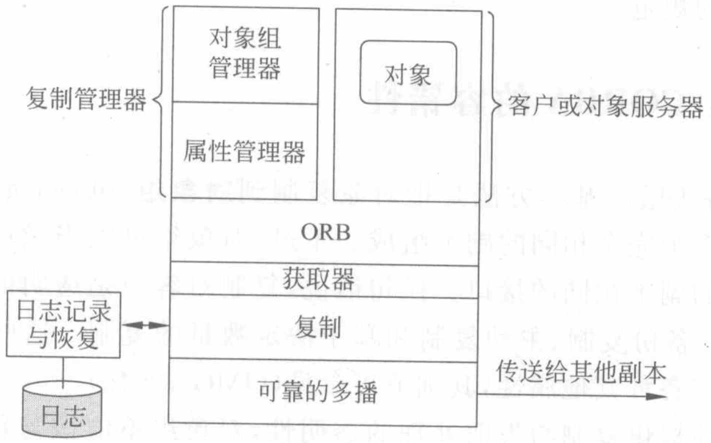


图 10.20 可容错的 CORBA 系统的示例体系结构

在此体系结构中，有几个扮演重要角色的组件。迄今为止最重要的是**复制管理器**(replication manager)，负责创建和管理一组复制的对象。从原理上讲，只有一个复制管理器，尽管出于容错性考虑复制管理器本身也可以复制。

如前所述，对客户来讲对象组和其他类型的 CORBA 对象没有根本的区别。要创建对象组，客户只需简单地调用提供的普通 create_object 操作。在这种情况下，由复制管理器指定所创建的对象类型。客户仍然不知道它正在隐式地创建一个对象组这一事实。开始创建一个新对象组时，创建的副本数量通常由与系统相关的默认值确定。复制管理器还负责在出现失败的情况下替换副本，以确保副本的数量不会低于指定的最小值。

此体系结构还展示了消息级截取器的使用。在 Eternal 系统中，每个调用都被截取并传递给单独的复制组件，这些组件维护对象组所要求的一致性并保证信息已被记录，从而可以恢复。

随后，调用请求通过可靠的、完全有序的多播发送给其他组成员。在主动复制的情况下，调用请求通过传递给对象的底层运行时系统，从而传递给每个对象副本。但是，在被动复制的情况下，调用请求仅仅传递给主版本的 RTS，而其他服务器只是记录调用请求以备恢复。当主机完成了调用时，就将其状态多播给备份。

这种体系结构基于使用截取器。还存在一些替代解决方案，包括那些在运行时系统中纳入了容错性的解决方案(可能会影响互操作性)，或者在 RTS 上面使用特殊服务来提供容错性的解决方案。除了这些区别之外，实际情况表明 CORBA 标准还没有涵盖到其他一些问题。举一个在实际中发生的问题，如果在不同的实现上创建副本，就无法保证这种方法会实际地工作。(Felber 和 Narasimhan 2004)中讨论了不同的方法，并且评价了 CORBA 中的容错性。

10.7.2 示例：Java 的容错性

考虑到 Java 作为一种用于开发分布式应用程序的语言和平台的普及性，目前做了一些工作在 Java 运行时系统中添加容错性。一个有趣的方法是确保 Java 虚拟机可用于主动复制。

主动复制实质上规定副本服务器像确定性状态机那样执行(Schneider 1990)。Java 中用于履行这种角色的极佳候选是 Java 虚拟机(Java Virtual Machine,JVM)。不幸的是,JVM 根本不是确定性的。有多种原因导致非确定性的行为,(Napper 等 2003)以及(Friedman 和 Kama 2003)中逐一描述了它们:

(1) JVM 可以执行本机代码,即 JVM 外部的代码,它们是通过一个接口提供给 JVM 的。JVM 把本机代码视作一个黑盒:它只能看到接口,但是对调用产生的(可能是非确定性的)行为一无所知。因此,为了把 JVM 用于主动复制,就必须确保本机代码以确定性的方式运行。

(2) 输入数据可能是非确定性的。例如,可以被多个线程操纵的共享变量可能为 JVM 的不同实例而改变,只要允许这些线程并发工作即可。为了控制这种行为,至少应该通过锁来保护共享数据。如所证实的那样,Java 运行时环境并非总是遵守这条规则,尽管它支持多线程。

(3) 在发生失败时,不同的 JVM 将产生不同的输出,表明机器已被复制。当 JVM 需要恢复到相同的状态中时,这种区别可能引发问题。如果可以假定所有的输出都是幂等(idempotent)的(即可以简单地重演),或者都是可测试的以便人们可以检查输出是否是在崩溃之前产生的,就可以简化问题。注意:必须做出这种假定,以便允许副本服务器决定它是否应该重新执行一个操作。

实践证明:把 JVM 转变成确定性的有限状态机决不是无足轻重的事情。需要解决的一个问题是:副本服务器可能崩溃这一事实。一种可能的组织结构是:让服务器依据主版本-副本模式运行。在这种模式中,一个服务器协调需要执行的所有动作,并且时不时地指示备份也这样做。当然,需要在主版本与备份之间进行小心的协调。

注意:尽管存在在主-备份设置中组织副本服务器这一事实,我们仍然需要处理主动复制:通过让每个副本以相同的顺序执行相同的操作,使它们保持最新。不过,为了确保所有的服务器都会执行相同的确定性的行为,应该把一个服务器的行为用作样本,让其他服务器遵照执行。

在这种设置中,(Friedman 和 Kama 2003)沿用的方法是:让主版本首先执行称为**帧**(frame)的指令。帧包括几个上下文切换(context switch),可能结束于线程因 I/O 完成而阻塞,或者是在发生了预定义次数的上下文切换之后。无论何时线程发出一个 I/O 操作,线程就会被 JVM 阻塞而停止运行。当一个帧启动时,主版本会让所有的 I/O 请求一个接一个地继续运行,并把结果发送给其他的副本。这样,至少会强制执行关于 I/O 操作的确定性的行为。

这种模式的问题显而易见:主版本总是位于其他副本之前。我们需要考虑两种情况。首先,如果非主版本的副本服务器崩溃,除了容错度会下降之外,不会造成真正的损害。另一方面,当主版本崩溃时,我们可能发现我们自身处在数据(或者操作)丢失的情形中。

为了把损失减至最小,主版本将在逐个帧的基础上工作。也就是说,仅当其当前帧完成之后,它才会把更新信息发送给其他副本。这种方法的作用是:当主版本在第 k 个帧上工作时,其他副本服务器将具有处理第 k 个帧前面的那个帧所需的所有信息。通过减少帧数可以限制所造成的损失,其代价是需要在主版本与备份之间进行更多的通信。

10.8 安全性

显然，安全性在任何分布式系统中都扮演着重要角色，在基于对象的分布式系统中也不例外。当考虑大多数基于对象的分布式系统时，分布式对象都是远程对象这一事实立即引出了如下情况：分布式系统的安全性体系结构都非常相似。实质上讲，如我们在第 9 章中所讨论的那样，通过标准的身份验证和授权机制来保护每个对象。

为了让人们清楚地看出安全性如何明确地适合于基于对象的分布式系统，我们将讨论 Globe 系统的安全性体系结构。如前所述，Globe 支持真正的分布式对象，其中单个对象的状态可以分散在多个机器上，并在它们当中进行复制。远程对象只是 Globe 对象的特例。因此，通过考虑 Globe 安全性体系结构，我们还可以看出如何将其方法同样好地应用于更传统的基于对象的分布式系统。在讨论了 Globe 之后，我们将简要探讨传统的基于对象的系统中的安全性。

10.8.1 示例：Globe

如我们所说过的，在少数几种基于对象的分布式系统中，一个对象的状态可以物理地分布在多个机器上并在它们当中进行复制，Globe 是其中之一。这种方法也引入了特定的安全性问题，它导致了(Popescu 等 2002)中描述的体系结构。

10.8.1.1 概述

当我们考虑在远程对象上调用一个方法的一般情况时，从安全性的角度讲至少有两个问题很重要：第一个问题是让调用者调用正确的对象；第二个问题是允许调用者调用该方法。我们把这两个问题分别称为**安全对象绑定**(secure object binding)和**安全方法调用**(secure method invocation)。对于 Globe 及其他支持复制或者到处移动对象的系统来说，我们还有另外一个问题，即**平台安全性**(platform security)。这种安全性包含两个问题。第一，对于复制对象的平台，如何保护它以阻止对象中包含的任何恶意代码；第二，如何保护对象以阻止恶意的副本服务器。

能够把对象复制到其他主机还引出了另一个问题。由于宿主对象副本的对象服务器并不需要总是完全可信的，就必须有一种机制来阻止任何宿主对象的副本服务器还被允许执行对象的任何方法。例如，对象的所有者可能希望限制只有一小组副本服务器能够执行更新方法，而只读取对象状态的方法则可能被任何经过身份验证的服务器执行。可以通过**反向访问控制**(reverse access control)来强制执行这样的策略，下面将更详细的讨论它。

Globe 中部署了几种机制用于建立安全性。首先，每个 Globe 对象都有一个关联的公钥/私钥对，称为**对象密钥**(object key)。其基本思想是：任何知道对象私钥的人都可以为用户和服务器设置访问策略。此外，每个副本还有一个关联的**副本密钥**(replica key)，它也被构造成公钥/私钥对。这个密钥对是由目前宿主特定副本的对象服务器生成的。如我们将看到的，副本密钥用于确保特定的副本是给定的分布式共享对象的一部分。最后，还假定

每个用户具有唯一的公钥/私钥对，称为**用户密钥**(user key)。

这些密钥用于以证书形式设置各种不同的访问权限。经由对象发放证书。证书有三种类型，如图 10.21 所示。**用户证书**(user certificate)与特定的用户相关联，精确指定用户可以调用哪些方法。最终，该证书包含一个位串 U，其长度与可供对象使用的方法数量相同。当且仅当允许用户调用方法 M_i 时，U[i]=1。同样，还有一个**副本证书**(replica certificate)，对于给定的副本服务器，该证书用于指定它可以执行哪些方法。它也有一个关联的位串，当且仅当允许服务器执行方法 M_i 时，R[i]=i。

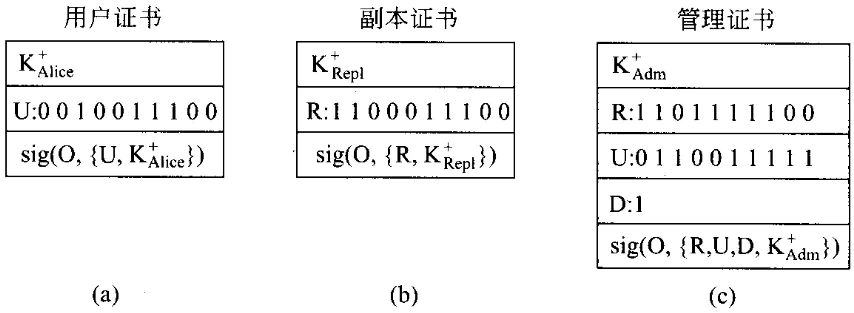


图 10.21 Globe 中的证书：
(a) 用户证书；(b) 副本证书；(c) 管理证书

例如，图 10.21(a)中的用户证书告诉 Alice(可以通过其公钥 K^+_{Alice} 标识)有权调用方法 M_2、M_5、M_6 和 M_7(注意：U 的索引开始于 0)。同样，副本证书指示拥有 K^+_{Repl} 的服务器允许执行方法 M_0、M_1、M_5、M_6 和 M_7。

管理证书可以由任何授权实体用于发放用户证书和副本证书。在这里，R 和 U 位串指定可以为哪些方法和哪些实体创建一个证书。此外，还有一个位指示管理实体是否可以把它的权限(的一部分)委托给别人。注意：当具有管理员角色的 Bob 为 Alice 创建一个用户证书时，他将用自己的签名(而不是对象的签名)来签署该证书。因此，Alice 的证书将需要回溯到 Bob 的管理证书，并最终回溯到用对象的私钥签署的管理证书。

在考虑可能大量地复制一些 Globe 对象时，管理证书很方便。例如，对象的所有者可能只想管理一组相对较少的永久副本，而把服务器初始副本的创建委托给宿主那些永久副本的服务器。在这种情况下，所有者可能决定允许永久副本安装所有用户具有只读权限的其他副本。无论何时 Alice 希望调用一个只读方法，她都会成功(假定她得到授权)。不过，当她希望调用一个更新方法时，她将不得不联系永久副本之一，因为其他的副本服务器都不允许执行这样的方法。

如我们所解释的，Globe 中的绑定过程要求将对象标识符(object identifier，OID)解析成联系地址。从原理上讲，任何支持平面名称的系统都可用于此目的。为了安全地把对象的公钥关联到其 OID 上，我们简单地把 OID 计算成公钥的一个 160 位的安全散列。这样，任何人都可以验证给定的公钥是否属于给定的 OID。这些标识符也称为**自我证明的名称**(self-certifying name)，它是最先在安全文件系统中提出的一个概念(Mazieres 等 1999)，我们将在第 11 章中讨论它。

我们还可以检查副本 R 是否属于对象 O。在这种情况下，我们只需要检查 R 的副本证书，以及检查谁发放它。签署者可能是一个具有管理权限的实体，在这种情况下，我们需要

检查其管理证书。底线是我们可以构造一个证书链，其中最后一个证书是使用对象的私钥签署的。在这种情况下，我们就知道R是O的一部分。

为了防止对象和主机相互之间发生干扰，部署了用于移动代码的技术，如第9章所述。可以利用特殊的审计技术来检测对象是否被篡改，我们将在第12章中讨论它。

10.8.1.2 安全方法调用

让我们现在深入研究安全地调用Globe对象的方法的细节。图10.22描述了从请求一个调用到实际地在副本上执行操作的完整过程。如图所示，总共需要依次执行13个步骤，下面描述了它们。

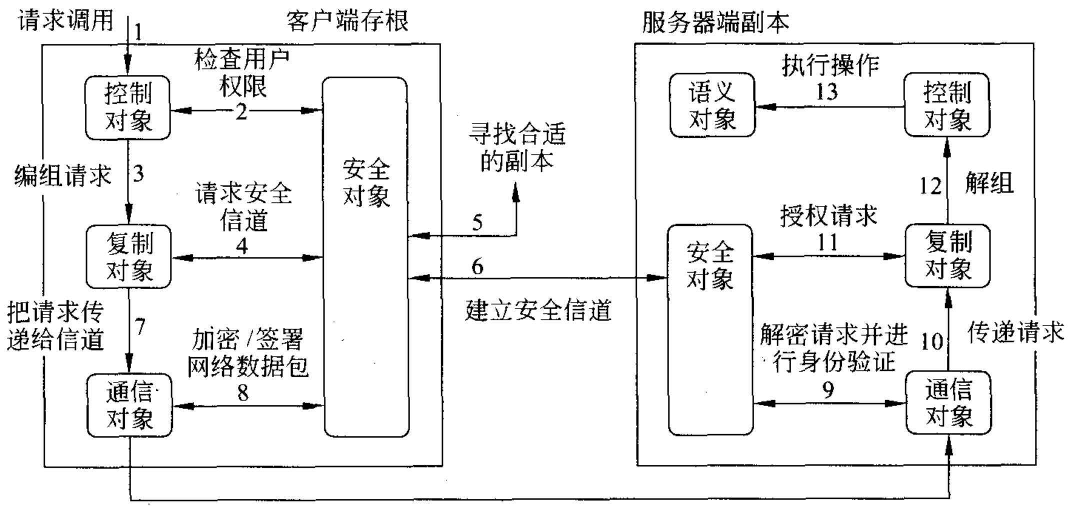


图10.22 Globe中的安全方法调用

(1) 首先，应用程序通过在本地调用关联的方法发出一个调用请求，就像在RPC中调用一个过程一样。

(2) 控制子对象利用存储在本地安全对象中的信息检查用户权限。在这种情况下，安全对象应该具有有效的用户证书。

(3) 编组并传递请求。

(4) 复制子对象请求中间件建立通往合适副本的安全信道。

(5) 安全对象首先启动副本查询。为了实现这个目标，它可以使用能够查找副本的任何命名服务，这些副本被指定能够执行某些方法。Globe定位服务被修改成处理这种查找(Ballintijn 2003)。

(6) 一旦找到合适的副本，安全子对象就可以与其对应的一方建立一条安全信道，然后把控制返回给复制子对象。注意：这个建立过程的一部分要求副本证明它被允许执行请求的调用。

(7) 现在把请求传递给通信子对象。

(8) 这个子对象加密并签署请求，使得可以通过信道传递它。

(9) 在接收到请求后，将对它进行解密和身份验证。

(10) 然后简单地把请求传递给服务器端复制子对象。

(11) 进行授权：在这种情况下，将把用户证书从客户端存根传递给副本，使得我们可

以验证请求确实被执行。

(12) 然后解组请求。

(13) 最后,可以执行操作。

尽管这看起来好像包含相对较多的步骤,但是这个示例显示了如何将一个安全方法调用分解成较小的单元,其中每个单元是确保经过身份验证的客户可以在经过身份验证的副本上执行授权调用所需要的。几乎所有基于对象的分布式系统都遵循这些步骤。它们与Globe之间的区别是:需要定位一个合适的副本,并且这个副本需要证明它可以执行方法调用。我们把这个证明作为习题留给读者完成。

10.8.2 远程对象的安全性

在使用远程对象时,我们经常看到对象引用本身被实现为完整的客户端存根,其中包含访问远程对象所需的所有信息。最简单形式的引用包含对象的准确联系地址,并且使用标准的编组和通信协议把调用传递给远程对象。

不过,在像Java这样的系统中,客户端存根(称为**代理**(proxy))可以是几乎任何事物。其基本思想是:远程对象的开发人员也开发代理,随后用目录服务注册代理。当客户寻找对象时,它最终将联系目录服务,获取代理并安装它。显然,这种方法有一些严重的问题。

第一,如果目录服务被劫持,那么攻击者也许就能够把伪造的代理返回给客户。实际上,这种代理可能危及客户与宿主远程对象的服务器之间的所有通信,从而给它们造成损失。

第二,客户无法对服务器进行身份验证:它只有一个代理,因此它与服务器之间进行的所有通信都必须经过寻个代理。这可能不是人们想要的情况,特别是由于客户现在只需要信任代理将正确地执行其工作。

另外,让服务器对客户进行身份验证可能更困难。当把敏感信息发送给客户时,身份验证可能是必要的。此外,由于客户身份验证现在依靠代理,还有可能出现如下情况:攻击者嗅探客户,导致损坏远程对象。

(Li 等 2004b)中描述了一种常规的安全性体系结构,它可用于使远程对象调用更安全。在他们的模型中,他们假定代理的确是由远程对象的开发人员提供的,并且是用目录服务注册的。Java RMI 中沿用了这种方法,Jini 中也是如此(Sun Microsystems 2005)。

要解决的第一个问题是对远程对象进行身份验证。在他们的解决方案中,Li 和 Mitchell 提出了一种包含两个步骤的方法。首先,由远程对象签署从目录服务下载的代理,从而允许客户验证其来源。代理反过来又将使用带有服务器身份验证的 TLS 对对象进行身份验证,如我们在第9章中所讨论的那样。注意:确保代理的确正确地对对象进行身份验证是对象开发人员的任务。客户将不得不依靠这种行为,但是由于它能够对代理进行身份验证,所以依靠对象身份验证与信任远程对象将正当地工作处于相同的级别上。

为了对客户进行身份验证,将使用单独的身份验证程序。当客户查找远程对象时,将把它定向到身份验证程序,并通过它下载一个**身份验证代理**(authentication proxy)。这是一个特殊的代理,它提供了一个接口,远程对象可以通过该接口对客户自身进行身份验证。如果这个身份验证成功,那么远程对象(或者实际上是它的对象服务器)将把实际的代理传递

给客户。注意：这种方法允许身份验证独立于实际的代理使用的协议，这被认为是一个重要的优点。

隔离开客户身份验证的另一个重要优点是：它现在可能把专用代理传递给客户。例如，可能只允许某些客户执行只读方法。在这种情况下，在进行身份验证之后，将把只提供了这些方法的代理交给客户，而不会交给它其他方法，从而可以构思更精细的访问控制。

本章小结

大多数基于对象的分布式系统都使用一种远程对象模型，其中由服务器宿主的对象允许远程客户执行方法调用。在许多情况下，这些对象是在运行时构造的，这实际上意味着它们的状态（可能还包括代码）是在客户执行远程调用时加载进对象服务器中的。Globe 系统支持真正的分布式共享对象。在这种情况下，对象的状态可能物理地分布在多个机器上，并在它们之间进行复制。

为了支持分布式对象，把功能与额外的功能属性（如容错性和可伸缩性）隔开是重要的。最终，开发了高级的对象服务器用于宿主对象。对象服务器给基本对象提供了许多服务，包括用于存储对象的工具，或者确保对传入的请求进行串行化。另一个重要作用是提供了一个外部世界的幻象，使得数据集合与操作数据的过程可以对应于对象的概念。通过对象适配器来实现这种作用。

当涉及到通信时，流行的方式是通过远程方法调用（RMI）调用一个对象，它与 RPC 非常相似。一个重要的区别是：分布式对象一般会提供一个系统级对象引用，允许进程从任何机器访问一个对象。全局对象引用解决了阻碍 RPC 的访问透明性的许多参数传递问题。

可以用许多不同的方式来实现这些对象引用，从精确说明可以在什么位置联系远程对象的简单的被动式数据结构，到需要被客户简单调用的便携式代码。后一种方法现在通常用于 Java RMI。

在大多数系统中没有提供特殊的措施来处理对象同步。一个重要的例外是处理同步的 Java 方法的方式：同步仅发生在运行在同一台机器上的客户之间。运行在不同机器上的客户需要采用特殊的同步措施。这些措施不是 Java 语言的一部分。

入口一致性是一种用于分布式对象的明显的一致性模型，在许多系统中（通常隐式地）支持它。显然，我们可以自然地为每个对象关联单独的锁。复制对象引出的问题之一是复制调用。由于对象倾向于被视作黑盒，所以这个问题更加显而易见。

基于对象的分布式系统中的容错性在很大程度上遵循用于其他分布式系统的方法。一个例外是：通过让 Java 虚拟机像确定性状态机那样运行，尝试使之具有容错性。然后，通过复制许多这样的机器，我们就获得了提供容错性的自然方式。

分布式对象的安全性涉及支持安全方法调用的思想。Globe 是一个全面的示例，它把这些调用推广到复制对象。如同它所证明的，有可能把策略与机制清楚地隔离开。对于身份验证和授权都是如此。需要特别注意要求客户从目录服务中下载代理的系统，Java 经常使用这类系统。

习　题

1. 我们区分过远程对象与分布式对象。它们的区别是什么？
2. 为什么用接口定义语言定义对象的接口是很有用的？
3. 分布式对象的中间件系统的一些实现完全基于动态方法调用。甚至静态调用也会被编译为动态调用。这种方法有什么好处？
4. 给出一个简单的协议概要，实现对象调用的至多一次(at most once)语义。
5. 在异步方法调用中，客户端和服务器端的对象应该是持久对象吗？
6. 在本书中，我们提及 CORBA 的异步方法调用的实现不会影响对象的服务器端实现。解释为什么会这样。
7. 给出一个示例，展示(粗心地)使用回调机制可能轻松地导致不想要的情况。
8. 一个对象有可能具有多个伺服对象吗？
9. 在具有 CORBA 对象引用的特定于系统的实现时仍然能够与其他基于 CORBA 的系统交换引用吗？
10. 我们怎样为安全的 Globe 对象对查找服务返回的联系地址进行身份验证？
11. CORBA 中的对象引用与 Globe 中的对象引用之间的关键区别是什么？
12. 考虑 Globe。概括一个简单的协议，通过它在用户代理(它可以访问 Alice 的私钥)与一个副本(我们可以肯定它能够执行给定的方法)之间建立一条安全的信道。
13. 给出对象引用的一个示例实现，它允许客户绑定到一个临时的远程对象。
14. Java 及其他语言支持异常，当错误发生时就会产生异常。如何在 RPC 和 RMI 中实现异常？
15. 如何把持久异步通信纳入到基于对远程对象进行 RMI 的通信模型中？
16. 考虑一种支持对象复制的基于对象的分布式系统，其中所有的方法调用都是完全有序的。另外，假定对象调用是原子的(例如，由于每个对象在调用时是自动锁定的)。这样的系统会提供入口一致性吗？顺序一致性呢？
17. 如本书中所述，描述一种用于处理复制调用的基于接收方的模式。

第11章

分布式文件系统

鉴于共享数据是分布式系统的基础，分布式文件系统构成了许多分布式应用程序的基础就不足为奇了。分布式文件系统允许多个进程在长时期内以一种安全、可靠的方式共享数据。所以，它们已被用作分布式系统和分布式应用程序的基础层。在本章中，我们把分布式文件系统看作通用分布式系统的模型。

11.1 体系结构

我们首先将探讨分布式文件系统的一般组织结构。大多数系统都是遵循传统的客户-服务器体系结构构建的，但是也存在完全分散式的解决方案。在下面，我们将探讨这两类组织结构。

11.1.1 客户-服务器体系结构

许多分布式文件系统都是遵照客户-服务器体系结构组织的，Sun Microsystems 的**网络文件系统**(Network File System，NFS)是其中部署最广泛的分布式文件系统之一，它用于基于 UNIX 的系统。我们将在这一整章中把 NFS 用作基于服务器的分布式文件系统的规范示例。特别是，我们将重点关注 NFSv3 和 NFSv4，前者是广泛应用的 NFS 版本 3 (Callaghan 2000)，后者是最新的版本 4(Shepler 等 2003)。我们还将讨论它们之间的区别。

NFS 背后的基本思想是：每个文件服务器都提供其本地文件系统的一个标准化的视图。换句话说，它不关心如何实现本地文件系统；每个 NFS 服务器都支持相同的模型。这种方法也用于其他的分布式文件系统。NFS 带有一个通信协议，该协议允许客户访问存储在一个服务器上的文件，从而允许异构进程集合共享公用的文件系统，其中的进程可能运行于不同的操作系统和机器上。

NFS 及类似系统的底层模型是**远程文件服务**(remote file service)的模型。这个模型为客户提供对远程服务器所管理的文件系统的透明访问。不过，客户通常不知道文件的实际位置。NFS 代之以为客户提供访问此文件系统的接口，此接口类似于传统的本地文件系统所提供的接口。特别是，客户仅被提供包含多种文件操作的接口，而服务器负责实现这些操作。因此，这一模型也被称为**远程访问模型**(remote access model)，如图 11.1(a)所示。

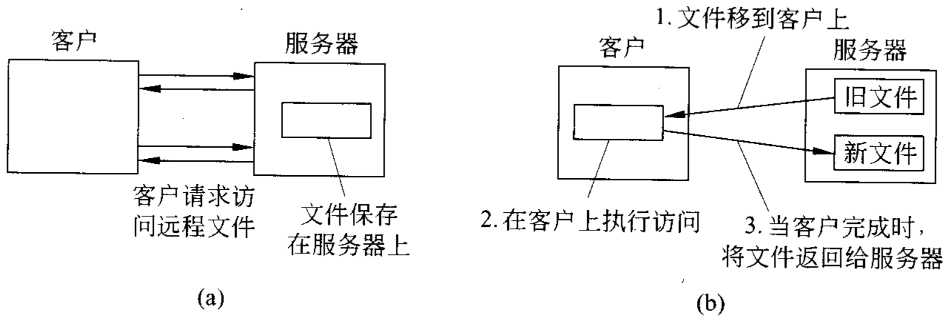


图　11.1

(a) 远程访问模型；(b) 上载/下载模型

与之相对照，在**上载/下载模型**(upload/download model)中，客户从服务器下载文件后，在本地访问该文件，如图 11.1(b)所示。客户完成对该文件的访问后，再将该文件上载回服务器，以便其他客户使用该文件。当客户下载一个完整的文件，修改该文件，然后将其放回服务器时，可以使用因特网的 FTP 服务。

尽管基于 UNIX 的 NFS 版本占主流地位，但是 NFS 已开发了多种版本以用于许多不同的操作系统。实际上，对所有现代 UNIX 系统而言，NFS 通常按照如图 11.2 所示的分层式体系结构实现。

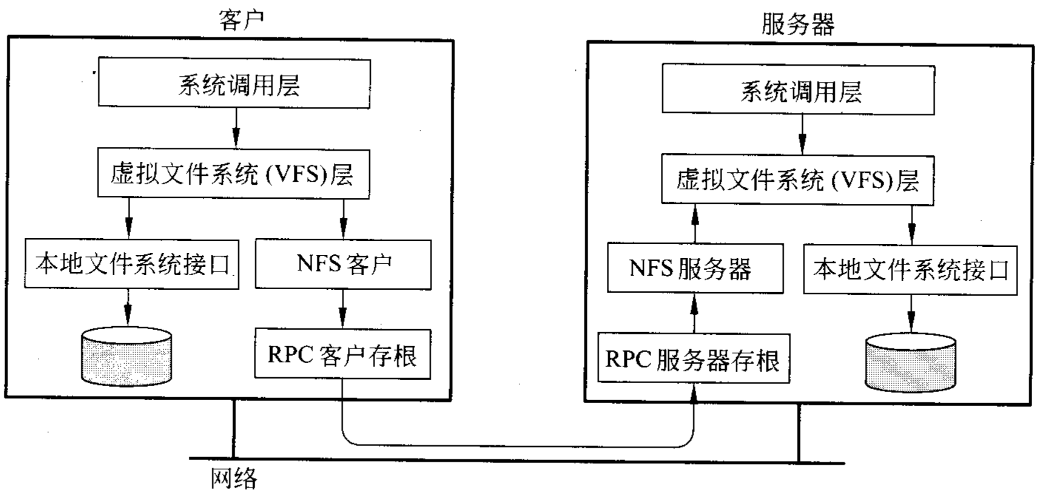


图 11.2　用于 UNIX 系统的基本 NFS 体系结构

客户使用其本地操作系统提供的系统调用访问文件系统。不过，本地 UNIX 文件系统接口已被**虚拟文件系统**(virtual file system，VFS)的接口代替。目前，VFS 已经成为不同(分布式)文件系统接口的事实上的标准(Kleiman 1986)。几乎所有的现代操作系统都提供了 VFS，如果不这样做，将或多或少地强制开发人员在采用新的文件系统结构时在很大程度上重新实现一个巨大的操作系统。利用 NFS，VFS 接口上的操作要么被传递给本地文件系统，要么被传递给一个称为 **NFS 客户**(NFS client)的单独组件，该组件负责处理对存储在远程服务器上的文件的访问。在 NFS 中，所有客户-服务器通信都是通过 RPC 完成的。NFS 客户将 NFS 文件系统的操作实现为服务器的 RPC。注意：VFS 接口所提供的操作可以不同于 NFS 客户所提供的那些操作。VFS 的整体思路是隐藏不同文件系统之间的差异。

在服务器端，我们可以看到相似的组织方式。**NFS服务器**(NFS server)负责处理传入的客户请求。RPC存根对请求进行解组，NFS服务器将它们转换成常规的VFS文件操作，随后将这些操作传递给VFS层。同样，VFS负责实现有储实际文件的本地文件系统。

这种模式的一个重要优点是NFS在很大程度上独立于本地文件系统。从原理上讲，它确实不关心客户端或服务器端的操作系统实现的是UNIX文件系统，还是Windows 2000文件系统，或者甚至是更老的MS-DOS文件系统。所关心的唯一重要问题是这些文件系统是否与NFS所提供的文件系统模型兼容。例如，因为MS-DOS使用短文件名，所以它不能用于以完全透明的方式实现NFS服务器。

文件系统模型

NFS所提供的文件系统模型几乎与基于UNIX的系统所提供的文件系统模型完全一样。文件被视为未解释的字节序列。它们被分层地组织到命名图中，图中的结点表示目录和文件。NFS像任何UNIX文件系统一样也支持硬链接和符号链接。文件具有文件名，但是访问它们是通过一种类似UNIX的**文件句柄**(file handle)实现的，我们将在下面详细讨论文件句柄。换句话说，为访问一个文件，客户必须先在命名服务中查找该文件的文件名，然后获得该文件关联的文件句柄。另外，每个文件都有许多属性，这些属性的值是可以查找和修改的。我们将在本章后面详细讨论文件命名。

图11.3分别列出了NFS版本3和版本4支持的常规文件操作。create操作用于创建一个文件，但是它在版本3和版本4中具有不同的含义。在版本3中，该操作用于创建常规

操作	版本3	版本4	描述
create	有	无	创建一个常规文件
create	无	有	创建一个非常规文件
link	有	有	创建一个文件的硬链接
symlink	有	无	创建一个文件的符号链接
mkdir	有	无	在给定目录下创建一个子目录
mknod	有	无	创建一个特殊文件
rename	有	有	更改文件名
remove	有	有	从文件系统中删除一个文件
rmdir	有	无	从一个目录中删除一个空的子目录
open	无	有	打开一个文件
close	无	有	关闭一个文件
lookup	有	有	根据文件名查找文件
readdir	有	有	读取一个目录下的项目
readlink	有	有	读取符号链接中所存储的路径名
getattr	有	有	获取文件的属性值
setattr	有	有	设置文件的一个或多个属性值
read	有	有	读取文件中包含的数据
write	有	有	向文件中写入数据

图11.3 NFS所支持的文件系统操作的不完全列表

文件。特殊的文件是使用单独的操作创建的。link 操作用于创建硬链接，symlink 用于创建符号链接。mkdir 用于创建子目录。诸如设备文件、套接字和命名管道之类的特殊文件是通过 mknod 操作创建的。

版本 4 完全改变了这种情况。在版本 4 中，create 用于创建**非常规**(nonregular)文件，这些非常规文件包括符号链接、目录和特殊文件。硬链接仍使用一个单独的 link 操作创建，但常规文件是通过 open 操作创建的，该操作是 NFS 中新增的操作，它是 NFS 版本 4 与老版本在文件处理方法上的主要差别。在版本 4 之前，NFS 被设计成允许它的文件服务器是无状态的。出于某些原因，NFS 版本 4 舍弃了这一设计标准。我们将在本章后面讨论这些原因。版本 4 假设服务器通常会维护同一文件上多个操作之间的状态。

操作 rename 用于更改现有文件的文件名，这与 UNIX 中的相同。

删除文件是通过 remove 操作实现的。在版本 4 中，这一操作用于删除任何类型的文件。而以前的版本需要使用一个单独的 rmdir 操作删除子目录。删除文件是通过文件名进行的，其效果是将该文件的硬链接个数减 1。如果其链接个数降为 0，那么该文件可能被破坏。

版本 4 允许客户打开和关闭(常规)文件。打开一个不存在的文件将导致创建一个新文件。要打开一个文件，客户需要提供文件名以及各种属性值。例如，客户可能指定以写访问方式打开文件。一个文件被成功打开后，客户可以通过该文件的文件句柄访问该文件。这个文件句柄也用于关闭该文件，客户通过这个文件句柄通知服务器，它不再需要访问该文件。然后，服务器可以释放它为客户访问该文件所保留的状态。

lookup 操作用于查找给定路径名的文件句柄。在 NFS 版本 3 中，lookup 操作不会越过挂载点(mount point)解析名称(回忆一下第 4 章的内容，挂载点是一个目录，该目录本质上代表指向**外部名称空间**(foreign name space)中的一个子目录的链接)。例如，假设名称/remote/vu 代表命名图中的一个挂载点。解析名称/remote/vu/mbox 时，NFS 版本 3 中的 lookup 操作将返回挂载点/remote/vu 的文件句柄，以及路径名中的剩余部分(即 mbox)。然后，客户必须显式地挂载完成名称查找所需的文件系统。这里所说的文件系统是一些文件、属性、目录和数据块的集合，它们共同地被实现为一个逻辑块设备(Tanenbaum 和 Woodhull 2006)。

版本 4 简化了这一操作。在版本 4 中，lookup 将试图解析整个名称，即使此名称越过挂载点。注意，这一方法只有在文件系统已经被挂载到挂载点时才是可行的。客户通过检查 lookup 完成时所返回的文件系统标识符，能够检测到挂载点已被越过。

读取目录中的项目有一个单独的操作 readdir。该操作返回一个(文件名，文件句柄)对的列表，以及客户请求的属性值。客户也可以指定返回的项目的个数。该操作返回一个偏移量，随后的 readdir 调用可以使用这个偏移量读取后面的项目。

操作 readlink 用于读取符号链接所关联的数据。通常，这一数据对应于随后查询文件的路径名。注意，lookup 操作不能处理符号链接。因而，遇到符号链接时，名称解析将停止，客户必须先调用 readlink，确定继续进行名称解析的位置。

文件具有多种关联的属性。同样，NFS 版本 3 和版本 4 在这一方面也有很大的不同，我们将在后面讨论它们的区别。典型的属性包括文件的类型(表明我们所处理的是目录、符号链接、特殊文件等)、文件长度、包含文件的文件系统的标识符，以及最后一次修改文件的

时间。操作 getattr 和 setattr 分别用于读取和设置文件属性。

最后，NFS 还有从文件读取数据和向文件写入数据的操作。通过 read 操作读取数据十分直观。客户可以指定将要读取的偏移量和字节数。该操作向客户返回已被读取的实际字节数，以及附加的状态信息（例如，是否到达文件末尾）。

向文件中写入数据是通过操作 write 完成的。同样，客户指定在文件中开始写入的位置、将要写入的字节数以及数据。另外，它可以指示服务器确保将所有数据都写入稳定存储器中（我们在第 8 章中曾讨论过稳定存储器）。NFS 服务器必须支持可以抵御电源故障、操作系统故障和硬件故障的存储设备。

11.1.2 基于群集的分布式文件系统

NFS 是许多分布式文件系统的一个典型示例，这些系统一般是依据传统的客户-服务器体系结构组织的。这种体系结构通常被增强用于服务器群集，此时，它将具有几个区别。

考虑通常用于并行应用程序的服务器群集，它们关联的文件系统将会做相应的调整，这并不令人奇怪。一种著名的技术是部署**文件带区划分技术**（file-striping technique），通过该技术可以将一个文件分布在多个服务器上。其基本思想很简单：通过把一个大文件分布在多个服务器上，就有可能并行地获取不同的部分。当然，仅当以并行数据访问有意义的方式组织应用程序时，这种组织方式才会工作得很好。这一般要求存储在文件中的数据具有非常规则的结构，例如，一个（稠密）矩阵。

对于通用的应用程序，或者那些具有不规则或者许多不同类型的数据结构的文件，文件带区划分可能不是一种有效的工具。在这些情况下，更方便的做法是：把文件系统作为一个整体进行分区并且简单地把不同的文件存储在不同的服务器上，而不是把单个文件划分到多个服务器上。图 11.4 中显示了这两种方法之间的区别。

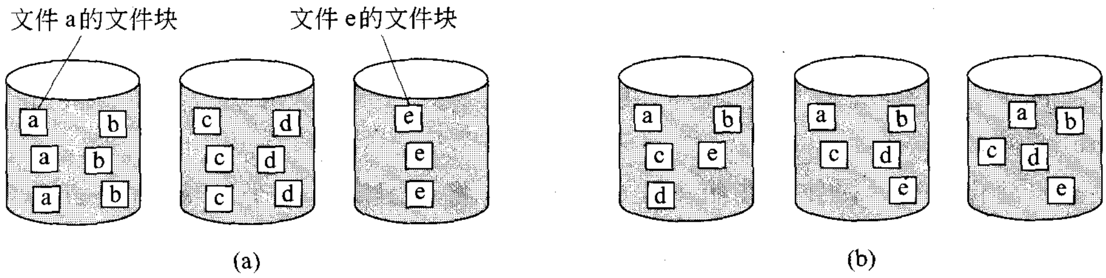


图 11.4 用于分布文件的两种方法之间的区别：
（a）把整个文件分布在多个服务器上；（b）对文件进行带区划分以进行并行访问

更有趣的是为特大型数据中心（如那些由像 Amazon 和 Google 这样的公司使用的数据中心）组织分布式文件系统。这些公司给 Web 客户提供服务，从而导致读取和更新分布在成千上万台计算机上的大量文件（Barroso 等 2003）。在这种环境下，关于分布式文件系统的传统假设不再有效。例如，我们可以想到在任何一个时刻都会有计算机出故障。

为了解决这些问题，例如，Google 开发了它自己的 Google 文件系统（Google file system，GFS），在（Ghemawat 等 2003）中描述了它的设计。Google 文件倾向于非常大，通常高达几 GB，其中每个文件都包含许多较小的对象。此外，通常通过追加数据而不是重写

文件的某些部分来更新这些文件。这些情况以及服务器会出故障的事实都是正常的，而不是例外，这导致构造的服务器群集如图 11.5 所示。

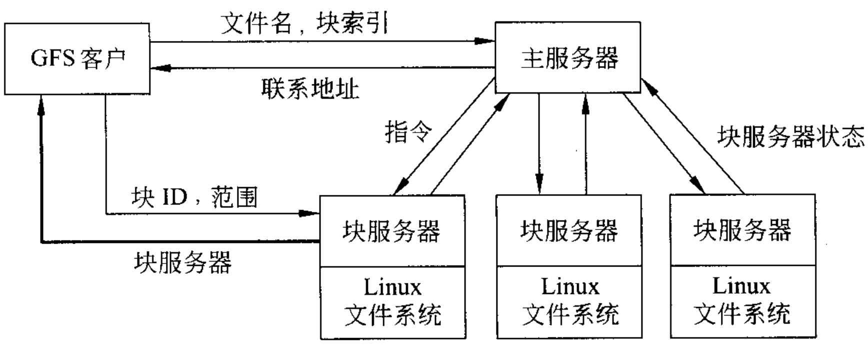


图 11.5　Google 服务器群集的组织结构

每个 GFS 群集都由一个主服务器以及多个**块服务器**(chunk server)组成。每个 GFS 文件都被分成多个块，其中每个块的大小为 64 MB，然后把这些块分布在所谓的块服务器上。要注意的重要一点是：仅当需要元数据信息时才联系 GFS 主服务器。特别是，GFS 客户把文件名和块索引传递给主服务器，并期望得到块的联系地址。联系地址包含访问正确的块服务器以获得所需文件块而需要的所有信息。

最终，GFS 主服务器实质上维护了一个名称空间，以及从文件名到块的映射。每个块都有一个关联的标识符，允许块服务器查找它。此外，主服务器会记录块所在的位置。块将被复制以处理失败，但是并不仅限于此。一个有趣的特性是：GFS 主服务器不会尝试保存块位置的准确记录。它会代之以偶尔联系块服务器来查看它们存储了哪些块。

这种模式的优点是比较简单。注意：主服务器可以控制把块分配给块服务器。此外，块服务器会保存它们存储了哪些块的记录。因此，一旦主服务器获得了块位置，它就具有数据存储在什么位置的准确视图。不过，如果这个视图必须在任何时间都保持一致，那么主服务器将变得很复杂。例如，每当块服务器崩溃或者添加服务器时，将需要通知主服务器。作为替代，通过轮询从当前的块服务器集刷新其信息要简单得多。GFS 客户只需知道主服务器相信哪些块服务器正在存储所请求的数据。由于块无论如何都会被复制，所以很有可能在至少一个块服务器上获得块。

这种模式为什么是可伸缩的呢？一个重要的设计问题是：主服务器在很大程度上处于控制之下，但是它不会构成瓶颈，这是由于它所做的所有工作决定的。采用了两种重要的措施来提供可伸缩性。

第一个(也是迄今为止最重要的)措施是：大量的实际工作是由块服务器完成的。当客户需要访问数据时，它会联系主服务器，查明哪些块服务器保存有该数据。之后，它只会与这些块服务器通信。根据主-备份模式复制块。当客户执行更新操作时，它会联系最近的保存该数据的块服务器，并把它的更新推送给该服务器。这个服务器将把更新推送给下一个最近的保存该数据的块服务器，依此类推。一旦传播了所有的更新，客户将联系主块服务器，然后它将给更新操作分配一个序列号，并把它传递给备份。与此同时，主服务器会避开循环。

第二个措施是：使用简单的**单级**(single-level)表来实现用于文件的名称空间，在该表

中将路径名映射到元数据(比如传统文件系统中对应的结点)。此外,将在主存中保存这一整张表,以及文件到块的映射。对这些数据的更新将被记录到持久存储器中。当日志变得过大时,将创建检查点,通过它存储主存数据,并且要求它可以被立即映射回主存中。因此,极大地减小了 GFS 主服务器的 I/O 密集性。

这种组织方式允许单个主服务器控制数百个块服务器,对于单个群集来说,这个规模是相当大的。然后,通过把像 Google 这样的服务组织进映射到群集上的较小服务中,不难想象可以使大量的群集集合协同工作。

11.1.3 对称式体系结构

当然,还存在基于对等技术的完全对称式体系结构。所有目前的提议都使用用于分布数据的基于 DHT 的系统,它们与基于键的查找机制结合使用。一个重要的区别是:它们是在分布式存储层的顶部构建文件系统,还是把全部文件都存储在参与结点上。

第一类文件系统的一个示例是 Ivy,它是使用基于 Chord DHT 的系统构建的一种分布式文件系统。在(Muthitacharoen 等 2002)中描述了 Ivy。他们的系统实质上由三个单独的层组成,如图 11.6 所示。最底层是由提供基本分散式查找工具的 Chord 系统构成的。中间是完全分布式的面向块的存储器层。最后,顶部有一个层实现类 NFS 文件系统。

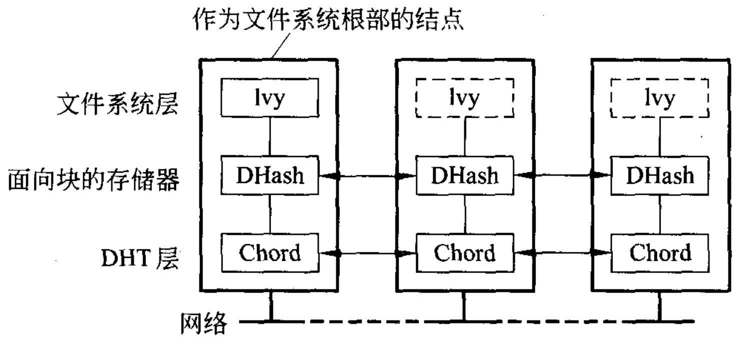


图 11.6 Ivy 分布式文件系统的组织结构

Ivy 中的数据存储是通过一种基于 Chord 的面向块的分布式系统实现的,该系统称为 **DHash**(Dabek 等 2001)。DHash 实质上相当简单。它只知道数据块,每个块的大小通常是 8 KB。Ivy 使用两种数据块。内容散列块(content-hash block)具有关联的密钥,它被计算为块内容的安全散列。这样,无论何时查找一个块,客户都可以立即验证是查找到了正确的块,还是返回了另一个块或者损坏的版本。此外,Ivy 还使用**公钥块**(public-key block),这种块把公钥用作查找密钥,并用关联的私钥签署了其内容。

为了提高可用性,DHash 把每个块 B 复制到负责存储 B 的服务器的 k 个直接后继中。此外,还会沿着查找请求经过的路由缓存查找到的块。

在 DHash 上面把文件实现为单独的数据结构。为了实现这个目标,每个用户都维护在文件上执行的操作的日志。出于简单起见,我们假定每个结点只有一个用户,使得每个结点将具有它自己的日志。日志是不可变记录的链表,其中每个记录都包含与 Ivy 文件系统上的某个操作相关的所有信息。每个结点只把记录追加到它自己的本地日志中。只有日志的

头部是可变的，并且指向最近追加的记录。每个记录都存储在单独的内容散列块中，而日志的头部则保存在公钥块中。

记录有多种不同的类型，它们粗略对应于 NFS 支持的不同操作。例如，当对文件执行更新操作时，就会创建一个**写**(write)记录，其中包含文件的标识符以及文件指针的偏移量和将要写入的数据。另外，还有用于创建文件(即添加新的索引**结点**(inode))、操纵目录等的记录。

为了创建一个新的文件系统，一个结点将简单地创建一个新的日志，以及一个将充当根的新索引结点。Ivy 会部署一个所谓的 **NFS 回环服务器**(NFS loopback server)，它只是一个本地用户级服务器，接受来自本地客户的 NFS 请求。就 Ivy 来说，这个 NFS 服务器支持挂载最近创建的文件系统，从而允许应用程序访问它，就像访问其他任何 NFS 文件系统一样。

在执行 read 操作时，本地 Ivy NFS 服务器将在日志上创建一个通行证，从表示相同数据块上的 write 操作的那些记录中收集数据，并且允许它获取最新存储的值。注意：由于每个记录都被存储为 DHash 块，可能需要执行**跨覆盖网络**(overlay network)的多个查找，以获取相关的值。

替代设计将不会使用单独的面向块的存储层，而是提议分布整个文件(而不是数据块)。Kosha(Butt 等 2004)的开发人员提议在特定的目录级别分布文件。在他们的方法中，每个结点都有一个名为/kosha 的挂载点，其中包含将使用基于 DHT 的系统分布的文件。在目录级别 1 上分布文件意味着位于子目录/kosha/a 中的所有文件都将存储在相同的结点中。同样，级别 2 上的分布意味着位于子目录. kosha/a/aa/中的所有文件都存储在相同的结点中。举一个级别 1 分布的例子，通过计算 a 的散列并把它用作查找中的密钥，可以找到负责把文件存储在/kosha/a 下面的结点。

这种方法的潜在缺点是：一个结点可能用光磁盘空间，来存储它负责的子目录中包含的所有文件。此外，一种简单的解决方案是：把那个子目录的一个分支放在另一个结点上，并且创建一个符号链接，指向现在存储该分支的位置。

11.2 进　程

在涉及到进程时，分布式文件系统并不具有什么与众不同的属性。在许多情况下，将会有不同类型的协作进程：存储服务器和文件管理器，就像我们在上面针对多种组织方式所描述的那样。

关于文件系统进程最有趣的方面是：它们是否应该是无状态的。NFS 是用于阐释折衷的良好示例。它长期独有的特性之一(与其他分布式文件系统相比)是服务器可以是无状态的。换句话说，NFS 协议不需要服务器维护任何客户状态。在版本 2 和版本 3 中沿用了这种方法，但是版本 4 废弃了它。

无状态方法的主要优点是比较简单。例如，当无状态服务器崩溃时，实质上不需要进入恢复阶段以把服务器带回以前的状态。不过，如我们在第 8 章中所解释的，我们仍然需要考虑：对于是否会准确地执行请求，我们并不能给客户提供任何保证。

在实际的实现中，并不能总是完全遵循 NFS 协议中的无状态方法。例如，无状态服务器不能轻松地锁定一个文件。就 NFS 来说，将使用单独的锁管理器来处理这种情况。此外，某些身份认证协议要求服务器维护其客户的状态。然而，在设计 NFS 服务器时，一般可能只需要维护客户的少量信息。一般来讲，这种模式足以胜任。

从版本 4 起，就废弃了无状态方法，尽管设计的新协议要求服务器维护关于其客户的大量信息。除了刚才提到的那些原因之外，还有其他原因要求选择有状态方法。一个重要的原因是：人们期望 NFS 版本 4 还能跨广域网工作。这要求客户可以有效地利用缓存，这反过来又需要一种有效的缓存一致性协议。这些协议通常在与能够保留客户所用文件的某些信息的服务器合作时工作得最好。例如，服务器可能把一个租约与它交给客户的每个文件相关联，并承诺给客户提供排他的读、写访问权限，直到租约到期或者被更新为止。我们将在本章后面回过头来讨论这些问题。

版本 4 与以前版本之间最明显的区别是对 open 操作的支持。此外，NFS 还支持回调过程，服务器可以通过它来执行对客户的 RPC。显然，回调还要求服务器记录其客户。

类似的推理影响了其他分布式文件系统的设计。总的来说，事实证明：维持完全的无状态设计可能十分困难，这通常导致构建有状态解决方案作为一种增强功能，就像 NFS 文件锁定那样。

11.3 通　　信

与进程一样，分布式文件系统中的通信没有什么特殊或与众不同之处。其中许多通信基于远程过程调用（RPC），尽管添加了一些有趣的增强特性来支持特殊情况。选择 RPC 机制的主要原因是使系统独立于底层操作系统、网络和传输协议。

11.3.1 NFS 中的 RPC

例如，在 NFS 中，客户与服务器之间的所有通信都遵循**开放式网络计算 RPC**（open network computing RPC，ONC RPC）协议以及表示编组数据的标准（Srinivasan 1995b）。（Srinivasan 1995a）中正式定义了开放式网络计算 RPC 协议。ONC RPC 类似于第 4 章中讨论的其他 RPC 系统。

每个 NFS 操作都可以被实现为对文件服务器的单一远程过程调用。事实上，在 NFS 版本 4 之前，客户通过保持请求相对简单来尽可能地简化服务器的工作。例如，客户为了第一次从一个文件中读取数据，通常必须先使用 lookup 操作查找文件句柄，然后它才能发出一个 read 请求，如图 11.7(a)所示。

这种方法需要两个连续的 RPC。如果在广域系统中使用 NFS，这种方法显然存在缺点。此时，第二个 RPC 的额外延迟可能导致性能降级。为了解决这些问题，NFS 版本 4 支持**复合过程**（compound procedure），可以通过它把多个 RPC 组合成单个请求，如图 11.7(b)所示。

在本示例中，客户将 lookup 和 read 请求组合成单个 RPC。在版本 4 中，在读文件之前

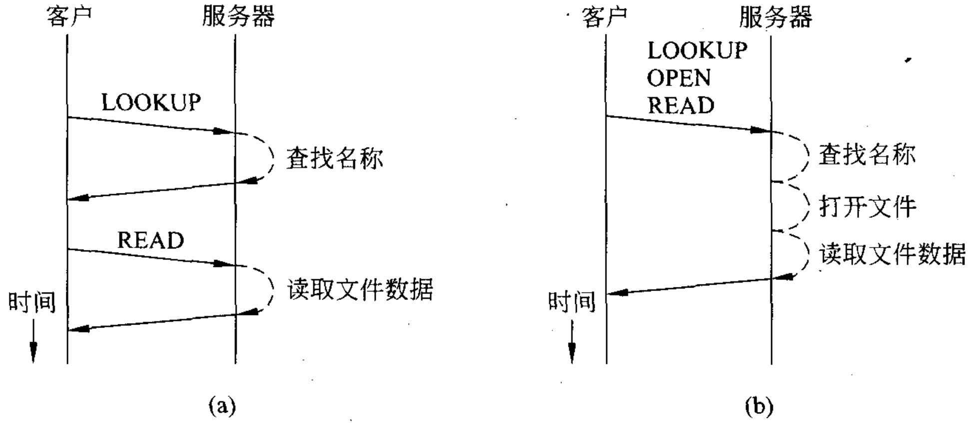


图 11.7

(a) 在 NFS 版本 3 中,从文件中读取数据;(b) 在 NFS 版本 4 中,使用复合过程读取数据

必须打开文件。在查找到文件句柄之后,将其传递给 open 操作,此后,服务器继续执行 read 操作。本示例的大体意思是:客户与服务器之间只需交换两条消息。

没有与复合过程关联的事务语义。组合在一个复合过程中的多个操作只是按照请求这些操作的顺序进行处理。如果存在来自其他客户的并发操作,那么它不会采取任何措施避免冲突。无论什么原因导致一个操作失败,复合过程中的后续操作都将不能执行,而只会把迄今为止所发现的结果返回给客户。例如,如果 lookup 失败,那么根本不会尝试执行后续的 open 操作。

11.3.2 RPC2 子系统

RPC 的另一个有趣的增强功能是作为 Coda 文件系统的一部分开发的(Kistler 和 Satyanarayanan,1992)。RPC2 是一个程序包,在(不可靠的)UDP 协议之上提供可靠的 RPC。每当调用远程过程时,RPC2 客户结点启动一个新的线程,该线程向服务器发送一个调用请求,随后阻塞,直到它接收到服务器的响应。因为请求处理可能需要在任意时间完成,所以服务器会定期向客户返回消息以通知客户它仍在处理请求。如果服务器死机,那么这个线程迟早会注意到这些消息停止了,并会向调用应用程序报告故障。

RPC2 的一个有趣方面是它对副作用的支持。**副作用**(side effect)是客户和服务器可以使用针对应用程序的协议进行通信的机制。例如,以客户打开视频服务器上的文件为例。本示例需要客户和服务器使用一种异步传输模式建立一个连续的数据流。也就是说,确保从服务器到客户的数据传输在最小的和最大的端到端延迟内完成。

RPC2 允许客户和服务器建立单独的连接以按时将视频数据传输给客户。连接建立是作为一个服务器的 RPC 调用的副作用完成的。为此,RPC2 运行时系统提供副作用例程的接口,该例程由应用程序开发人员实现。例如,有用于建立连接的例程和传输数据的例程。客户和服务器上的 RPC2 运行时系统分别自动调用这些例程,但是它们的实现却完全独立于 RPC2。图 11.8 显示了副作用的这一原理。

RPC2 的另一个不同于其他 RPC 系统的特性是它支持多播。Coda 的一个重要设计问题是服务器需要记录哪些客户具有文件的本地副本。当文件被修改时,服务器通过 RPC 通

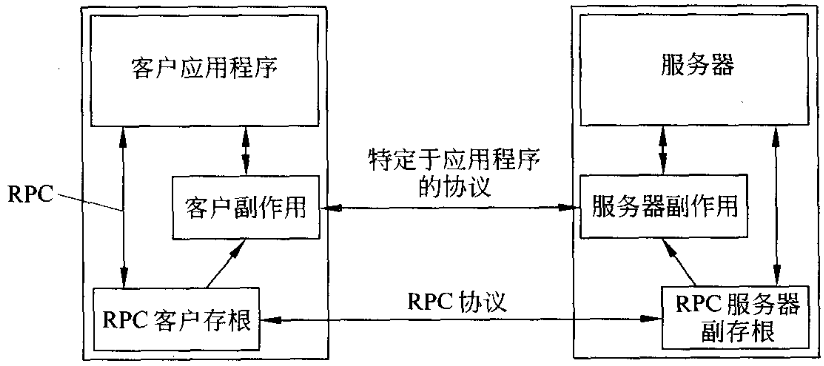


图 11.8　Coda 的 RPC2 系统中的副作用

知适当的客户，使这些文件的本地副本无效。显然，如果服务器一次只能通知一个客户，那么使所有的客户无效可能需要花一些时间，如图 11.9(a)所示。

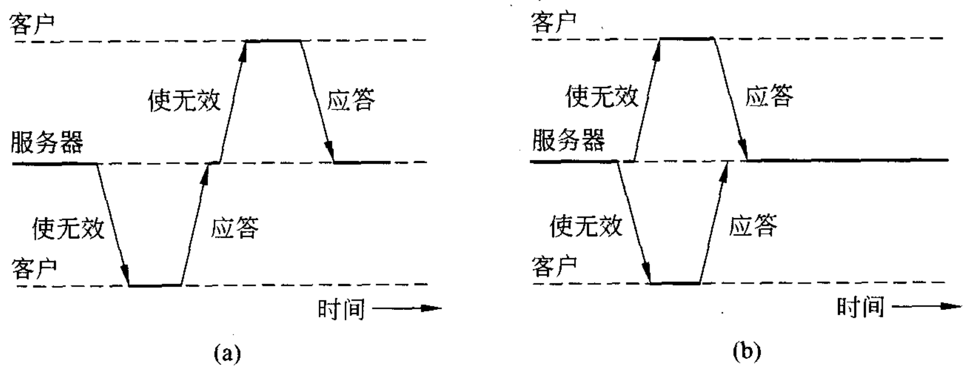


图　11.9

(a) 一次发送一条无效消息；(b) 并行地发送无效消息

这个问题是由 RPC 可能失败这样的事实引起的。按严格的顺序使文件无效可能会被延迟相当长的时间，这是因为服务器不可能到达一个可能已崩溃的客户，但是服务器只有在经过相对较长的时间期限后才会放弃那个客户。同时，其他客户仍会从它们的本地副本读取数据。

图 11.9(b)显示了一个替代(更好的)解决方案。服务器不是一个接一个地使每个副本无效，而是同时向所有客户发送一条无效消息。因此，所有无故障的客户都将在与执行立即 RPC 的相同时间内得到通知。此外，服务器会在通常的时间期限内注意到某些客户无法响应这个 RPC，从而能够宣布这样的客户已崩溃。

并行 RPC 是通过 **MultiRPC** 系统实现的，MultiRPC 系统是 RPC2 程序包的一部分(Satyanarayanan 和 Siegel 1990)。MultiRPC 的一个重要方面是：RPC 的并行调用对于被调用者而言是完全透明的。也就是说，MultiRPC 调用的接收者无法区分并行 RPC 和普通 RPC。对于调用者而言，并行执行在很大程度上也是透明的。例如，在出现故障时，MultiRPC 的语义与普通 RPC 的语义非常相似。同样，副作用机制也可按照前面所述的方式使用。

MultiRPC 本质上是通过并行地执行多个 RPC 实现的。这意味着调用者显式地向每个接收者发送一个 RPC 请求。不过，调用者并不立即等待响应，而是在所有请求都被发送出去之后才阻塞。也就是说，调用者调用许多单向 RPC，然后阻塞，直至接收到所有无故障的

接收者的响应为止。在 MultiRPC 中并行执行 RPC 的另一种方法是建立多播组，并使用 IP 多播向所有组成员发送 RPC。

11.3.3 Plan 9 中面向文件的通信

最后，值得提及用于处理分布式文件系统中的通信的一种完全不同的方法，即 Plan 9 (Pike 等 1995)。与其说 Plan 9 是一个分布式文件系统，不如说它是一个**基于文件的分布式系统**(file-based distributed system)。所有资源(甚至包括诸如进程和网络接口之类的资源)都是以相同的方式访问的，即使用类似于文件的语法和操作进行访问。这种思想是从 UNIX 继承而来的，UNIX 也试图为各种资源提供类似于文件的接口，但是 Plan 9 更深入、更一致地利用了类似于文件的接口。举例说明，网络接口是由文件系统表示的，因此，它由一些特殊文件组成。这种方法类似于 UNIX，尽管 UNIX 中的网络接口是由文件(而不是文件系统)表示的(注意：这里所说的文件系统同样是指包含所有数据和元数据的逻辑块设备，这些数据和元数据是由文件集合组成的)。例如，在 Plan 9 中，单独的 TCP 连接是由子目录表示的，该子目录包含图 11.10 所示的文件。

文件	描述
ctl	用于写协议特有的控制命令
data	用于读、写数据
listen	用于接收传入的连接建立请求
local	提供连接的呼叫方的信息
remote	提供连接的另一方的信息
status	提供连接的当前状态的诊断信息

图 11.10 在 Plan 9 中与单一 TCP 连接关联的文件

文件 ctl 用于向连接发送控制命令。例如，为使用端口 23 与 IP 地址为 192.31.231.42 的机器建立 Telnet 会话，发送者要向文件 ctl 中写入文本字符串“connect 192.31.231.42!23”。而接收者应当已经在它自己的 ctl 文件中写入字符串“announce 23”，指示它可以接收传入的会话请求。

data 文件通过简单地执行 read 操作和 write 操作来交换数据。这些操作采用文件操作的通用 UNIX 语义。例如，为了向连接写入数据，进程简单地调用以下操作：

```
res= write(fd, buf, nbytes);
```

其中 fd 是打开数据文件时返回的文件描述符，buf 是指向缓冲区的指针，该缓冲区包含将要写入的数据，nbytes 是应从缓冲区中提取的字节数。该操作将返回实际写入的字节数，并将其存储在变量 res 中。

文件 listen 用于等待连接建立请求。进程宣布它愿意接受新连接后，它可以在文件 listen 上执行阻塞的 read 操作。如果一个请求进入，该调用就向一个新的 ctl 文件返回一个文件描述符，这个文件描述符对应于新创建的连接目录。

11.4 命　名

命名无可争议地在分布式文件系统中起着重要的作用。在几乎所有的情况下，都在分层式名称空间中组织名称，如我们在第5章中所讨论的那样。下面我们将再次以NFS为例，介绍在分布式文件系统中如何处理命名。

11.4.1 NFS中的命名

NFS命名模型下面的基本思想是：允许客户完全透明地访问服务器维护的远程文件系统。这种透明性是通过使客户能够在它自己的本地文件系统中挂载一个远程文件系统实现的，如图11.11所示。

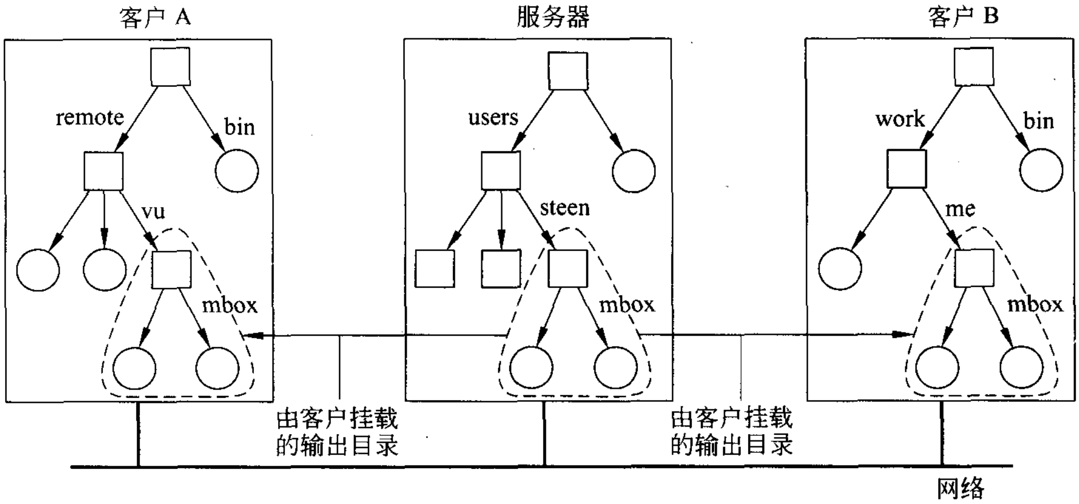


图11.11　在NFS中挂载远程文件系统（的一部分）

NFS允许客户不挂载整个文件系统，而只挂载文件系统的一部分，如图11.11所示。当服务器允许其他客户使用一个目录及该目录中的项目时，称该服务器**输出**(export)那个目录。一个输出目录可以由客户挂载到其本地名称空间中。

这种设计方法有一个重要的含义：从原理上讲，用户不共享名称空间。如图11.11所示，客户A上名称为/remote/vu/mbox的文件在客户B上被命名为/work/me/mbox。因此，一个文件的名称依赖于客户组织它们自己的本地名称空间的方式，以及挂载输出目录的位置。在分布式文件系统中，这种方法的缺点是使共享文件变得更困难。例如，Alice不能用她为某个文件指定的名称来告诉Bob这个文件，因为那个名称可能在Bob的文件名称空间中具有完全不同的含义。

解决这一问题的方法有很多种，但是最通用的方法是为每个客户提供一个部分标准化的名称空间。例如，每个客户都可以使用本地目录/usr/bin挂载一个包含标准程序集合的文件系统，该程序集合可供每个人使用。同样，目录/local也可用作挂载位于客户主机上的本地文件系统的标准目录。

一个 NFS 服务器自身可以挂载其他服务器输出的目录。但是，它不能再将这些目录输出给它自己的客户。作为替代，客户必须显式地从维护目录的服务器那里挂载该目录，如图 11.12 所示。加入这一限制的部分原因是出于简单性考虑。如果服务器可以输出它从其他服务器挂载的目录，那么它就可能返回包含服务器标识符的特殊文件句柄。NFS 不支持这种文件句柄。

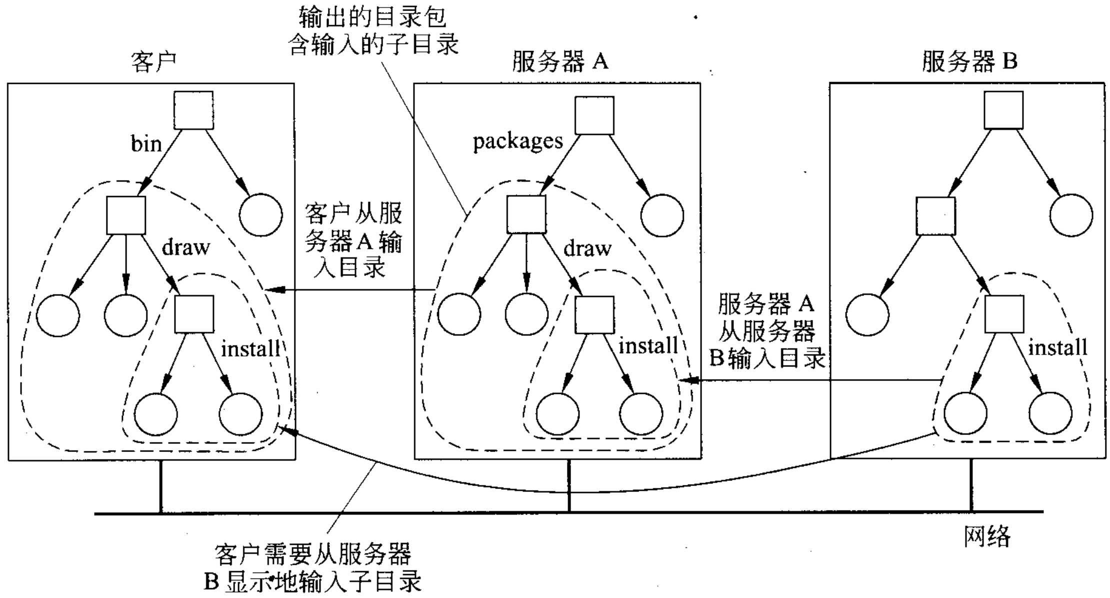


图 11.12　从 NFS 中的多个服务器挂载嵌套的目录

为了更详细地解释这一点，我们假设服务器 A 宿主文件系统 FS_A，并从该文件系统输出目录/packages。此目录包含子目录/draw，该子目录充当 A 挂载的文件系统 FS_B 的挂载点，文件系统 FS_B 是由服务器 B 输出的。假设 A 也向它自己的客户输出/packages/draw，并假设一个客户已经把/packages 挂载到它自己的本地目录/bin 中，如图 11.12 所示。

如果名称解析是迭代的(比如在 NFS 版本 3 中)，那么为了解析名称/bin/draw/install，当客户已经在本地解析了/bin 时，它将联系服务器 A，并请求 A 返回目录/draw 的文件句柄。此时，服务器 A 应该返回包含服务器 B 的标识符的文件句柄，因为只有 B 可以解析路径名的剩余部分，即本示例中的/install。如前所述，NFS 不支持这种类型的名称解析。

NFS 版本 3(以及更早版本)一次只能查找单个文件名，从这种意义上说，它的名称解析是严格迭代的。换句话说，解析诸如/bin/draw/install 之类的名称需要单独调用 NFS 服务器三次。此外，客户还完全负责实现路径名的解析。NFS 版本 4 还支持递归式名称查找。这样，客户可以将完整的路径名传递给服务器，并请求服务器解析它。

NFS 版本 4 还解决了 NFS 名称查找的另一个怪异问题。假设一个文件服务器宿主了多个文件系统。利用版本 3 的严格迭代式名称解析，每当 lookup 完成一个挂载其他文件系统的目录查找时，lookup 就会返回那个目录的文件句柄。随后读取那个目录时将返回该目录下的原始内容，而不是被挂载的文件系统的根目录下的内容。

为了解释这一问题，我们假设在先前的示例中，文件系统 FS_A 和 FS_B 是由单个服务器宿主的。如果客户已经将/packages 挂载到它的本地目录/bin 中，那么在该服务器上查找文件名 draw 将会返回 draw 的文件句柄。随后使用 readdir 请求该服务器列出的目录项目

时，将会返回存储在 FS_A 的子目录/packages/draw 下的原始目录项目列表。仅当客户也挂载了文件系统 FS_B 时，才可能正确地解析相对于/bin 的路径名 draw/install。

NFS 版本 4 通过允许查找跨越服务器上的挂载点来解决这一问题。特别是，lookup 返回被挂载目录的文件句柄，而不是原始目录的文件句柄。客户可以通过检查被查找文件的文件系统标识符来检测 lookup 是否跨越了挂载点。如果需要，客户也可以在本地挂载那个文件系统。

11.4.1.1 文件句柄

文件句柄是对文件系统内的文件的引用。它与所引用的文件的名称无关。文件句柄是由该文件系统所在的服务器创建的，并且它在该服务器输出的所有文件系统中是唯一的。它是在创建文件时创建的。客户不知道文件句柄的实际内容，它是完全不透明的。NFS 版本 2 的文件句柄长度是 32 字节，但是 NFS 版本 3 和版本 4 可具有可变长度的文件句柄，版本 3 的文件句柄的最大长度为 64 字节，版本 4 的文件句柄的最大长度为 128 字节。当然，文件句柄的长度是透明的。

在理想情况下，文件句柄被实现为一个与文件系统有关的文件的真正标识符。首先，这意味着只要文件是存在的，那么它就应该有一个不变的文件句柄。这种持久性需求允许客户只要通过文件名找到了关联的文件，就可以在本地存储文件句柄。一个好处是性能：因为许多文件操作需要使用文件句柄，而不使用文件名，所以客户就可以避免在每个文件操作之前反复地查找名称。这种方法的另一个好处是客户现在可以访问文件，而不依赖于文件的(当前)名称。

因为文件句柄可以被客户存储在本地，所以服务器不能重新使用被删除文件的文件句柄，这一点也是非常重要的。否则，当客户使用它在本地存储的文件句柄时，它可能错误地访问错误的文件。

注意：迭代式名称查找和不允许 lookup 操作跨越挂载点结合在一起就引出了一个如何获得初始文件句柄的问题。为了访问远程文件系统上的文件，客户需要向服务器提供应查找的目录的文件句柄，以及将要解析的文件或目录的名称。NFS 版本 3 使用一个单独的挂载协议解决这一问题，客户实际上通过这一协议挂载远程文件系统。挂载文件系统后，客户获得被挂载的文件系统的**根文件句柄**(root file handle)，随后可以使用这个根文件句柄作为查找名称的起点。

在 NFS 版本 4 中，这一问题是通过提供一个单独的操作 putrootfh 解决的，该操作通知服务器解析与它管理的文件系统的根文件句柄相关的所有文件名。根文件句柄可用于查找服务器的文件系统上的任何其他文件句柄。这种方法的额外好处是不需要使用单独的挂载协议。而可以代之以将文件系统的挂载集成到查找文件的常规协议之中。客户使用 putrootfh 获得文件系统的根文件句柄，通过请求服务器解析相对于这个根文件句柄的名称，就可以简单地挂载一个远程文件系统。

11.4.1.2 自动挂载

如前所述，NFS 命名模型本质上为各个用户提供了自己的名称空间。在这个模型中，如果各个用户对同一文件的命名不同，那么可能难以实现文件共享。这个问题的一种解决

方案是为每个用户提供一个部分标准化的本地名称空间，然后为每个用户以同样的方式挂载远程文件系统。

NFS 命名模型的另一个问题是应该何时挂载远程文件系统。以一个具有成千上万个用户的大型系统为例。假设每个用户都有一个本地目录/home，该目录用于挂载其他用户的主目录。例如，虽然 Alice 的实际文件存储在远程服务器上，但是她可以通过本地可用的/home/alice 访问她的主目录。这个目录可以在 Alice 登录到她的工作站时自动挂载。另外，她可能经由/home/bob 访问 Bob 的目录，从而访问 Bob 的公共文件。

但是，问题是 Bob 的主目录是否也应该在 Alice 登录时自动挂载。这种方法的好处是整个挂载文件系统的工作都是对 Alice 透明的。但是，如果每个用户都使用这一策略，那么登录可能导致通信和管理的许多额外开销。另外，它可能要求所有用户都是事先已知的。另一个更好的方法是：根据需要（也就是第一次需要使用时），透明地挂载其他用户的主目录。

在 NFS 中，根据需要挂载一个远程文件系统（或者实际上是一个输出目录）是由**自动挂载器**（automounter）处理的，它以一个单独进程的形式运行于客户机器上。自动挂载器的基本原理相对比较简单。我们来考虑一个作为 UNIX 操作系统上的用户级 NFS 服务器实现的简单自动挂载器。关于其他可选择的实现（Callaghan 2000）。

如上所述，假设每个用户可以通过本地目录/home 访问所有用户的主目录。客户端机器启动时，自动挂载器开始挂载这个目录。这个本地挂载的效果是：每当程序试图访问/home 时，UNIX 内核会把 lookup 操作转发给 NFS 客户，此时，NFS 客户将该请求转发给作用相当于 NFS 服务器的自动挂载器，如图 11.13 所示。

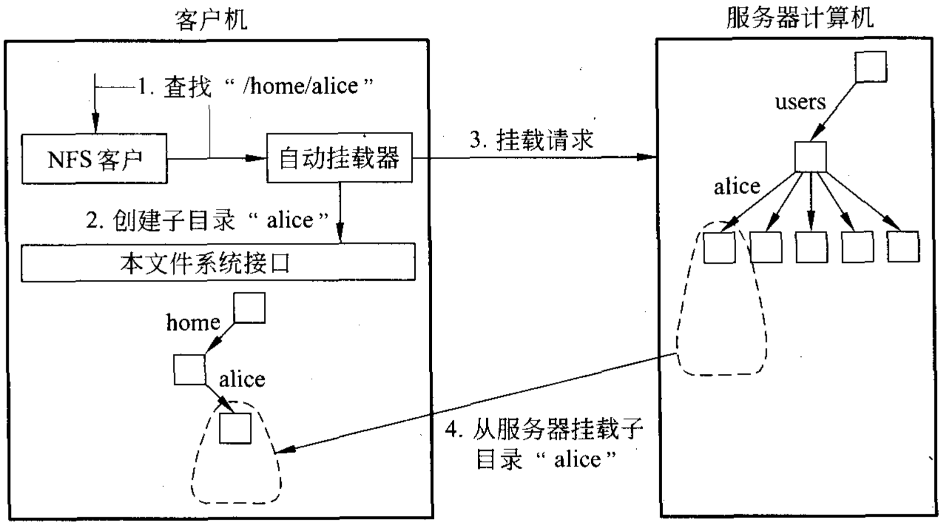


图 11.13 简单的 NFS 自动挂载器

例如，假设 Alice 登录进入系统。登录程序会试图读取目录/home/alice 以寻找诸如登录脚本之类的信息。因而，自动挂载器会接收到查找子目录/home/alice 的请求，为此它首先在/home 中创建子目录/alice。然后，它查找输出 Alice 的主目录的 NFS 服务器以便随后在/home/alice 中挂载那个目录。此时，登录程序可以继续执行。

这种方法的问题是自动挂载器将不得不参与所有的文件操作以保证透明性。如果一个

被引用的文件因为相应的文件系统还没有被挂载，从而不能在本地可用，那么自动挂载器将必须知道这一事实。特别是，它将需要处理所有的读请求和写请求，甚至对于已经挂载的文件系统也是如此。这种方法可能导致严重的性能问题。使自动挂载器只挂载和卸除目录，而在其他时候不参与循环可能会更好。

一个简单的解决方案是让自动挂载器在特殊的子目录中挂载目录，并为每个被挂载的目录安装一个符号链接。图 11.14 显示了这种方法。

这个示例将用户主目录挂载为/tmp_mnt 的子目录。当 Alice 登录时，自动挂载器在/tmp_mnt/home/alice 中安装她的主目录，并创建一个指向该子目录的符号链接/home/alice。这样，每当 Alice 执行如下一个命令时：

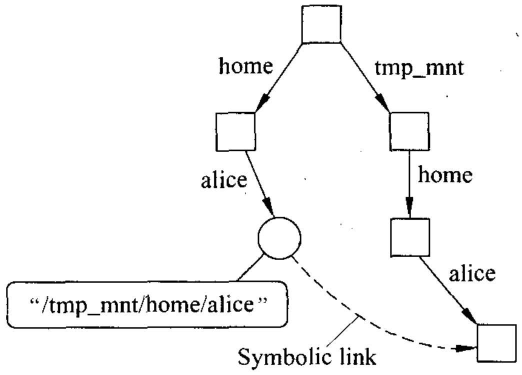


图 11.14 自动挂载时使用符号链接

```
ls  l /home/alice
```

系统将直接联系输出 Alice 的主目录的 NFS 服务器，而无需自动挂载器的进一步参与。

11.4.2 构造全局名称空间

通常通过把多种不同的遗留系统组合成一个整体来构造大型分布式系统。当需要提供对文件的共享访问时，至少要有一个全局名称空间。目前，通常可以使用基本的方法（比如通过 FTP 访问）来共享文件系统。例如，这种方法一般用在网格计算中。

真正的广域分布式文件系统使用的是更先进的方法，但是，在采用这些方法时，通常需要修改操作系统内核。因此，研究者在寻找一些方法把现有的文件系统集成进单个全局名称空间中，但是只使用用户级解决方案。（Anderson 等 2004）提出了这样一个系统，简称为**全局名称空间服务**（Global Name Space Service，GNS）。

GNS 没有提供访问文件的接口。作为替代，它只提供了建立全局名称空间的方法，该全局名称空间合并了现有的名称空间。最终，GNS 客户将维护一棵虚拟树，其中每个结点都是一个目录或者一个**接合点**（junction）。接合点是一个特殊的结点，它指示名称解析将由另一个进程接管，因此与传统文件系统中的挂载点具有一些相似之处。有 5 种不同类型的接合点，如图 11.15 所示。

接合点	描述
GNS接合点	引用另一个 GNS实例
逻辑文件系统名称	引用将在定位服务中查找的子树
逻辑文件名称	引用将在定位服务中查找的文件
物理文件系统名称	引用可直接远程访问的子树
物理文件名称	引用可直接远程访问的文件

图 11.15 GNS 中的接合点

GNS接合点简单地引用另一个GNS实例,它只是可能宿主在另一个进程上的另一棵虚拟树。两种逻辑接合点包含联系定位服务所需的信息。定位服务将提供分别用于访问文件系统和文件的联系地址。物理文件系统名称引用另一个服务器上的文件系统,并且在很大程度上对应于一个逻辑接合点将需要的联系地址。例如,像 ftp://ftp.cs.vu.nl/pub 这样的 URL 将包含访问指定的 FTP 服务器上的文件所需的所有信息。类似地,像 http://www.cs.vu.nl/index.htm 这样的 URL 是物理文件名称的典型示例。

显然,接合点应该包含继续进行名称解析所需的所有信息。有许多方式可用于执行这一任务,但是考虑到有如此之多的不同文件系统,每个特定的接合点都将需要它自己的实现。幸运的是,还有许多访问远程文件的常用方式,包括用于同 NFS 服务器、FTP 服务器以及基于 Windows 的机器(特别是 CIFS)进行通信的协议。

GNS的一个优点是:能够把文件的命名与其实际位置分离开。虚拟树与文件和目录所在的物理位置毫不相关。此外,通过使用定位服务,还可能到处移动文件,而不会使它们的名称无法解析。如果那样的话,就需要在定位服务中限制新的物理位置。注意:这与我们在第5章中所讨论的完全相同。

11.5 同　　步

现在我们将重点关注分布式文件系统中的同步问题。有多种问题需要我们注意。首先,如果不共享文件,那么文件系统的同步将不是一个问题。不过,在分布式系统中,当性能问题至关重要时,文件共享的语义将变得稍微复杂一点。最终,提出了一些不同的解决方案,我们接下来将讨论其中一些最重要的解决方案。

11.5.1 文件共享的语义

当两个或更多的用户同时共享同一文件时,为避免出现问题,精确定义读操作和写操作的语义是十分必要的。在允许进程共享文件的单处理器系统(如 UNIX)中,语义通常规定当 read 操作出现在 write 操作之后时,read 操作返回刚写入的值,如图 11.16(a)所示。类似地,当相继很快地发生两个 write 操作后又发生一个 read 操作时,那么所读取的值就是最后一个 write 操作写入的值。实际上,系统对所有操作都强加了一个绝对时间顺序,并且总是返回最新的值。我们称这个模型为 **UNIX 语义**(UNIX semantics)。这个模型易于理解,实现起来也相当简单。

在分布式系统中,只要只有一个文件服务器和客户不缓存文件,那么实现 UNIX 语义十分容易。所有 read 操作和 write 操作直接传递到文件服务器,该文件服务器以严格的顺序执行它们。这种方法给出了 UNIX 语义(只不过它有个小问题,那就是网络延迟可能导致落后于 write 操作 1 ms 发生的 read 操作先达到服务器,从而导致 read 操作得到旧值)。

不过,实际上,所有文件请求都必须传递给单个服务器的分布式系统的性能通常是很差的。通常,解决这个问题的方法是允许客户在它们的私有(本地)缓存中保留频繁使用的文件的本地副本。尽管我们下面将详细讨论文件缓存,但是我们现在可以指出,如果客户在本

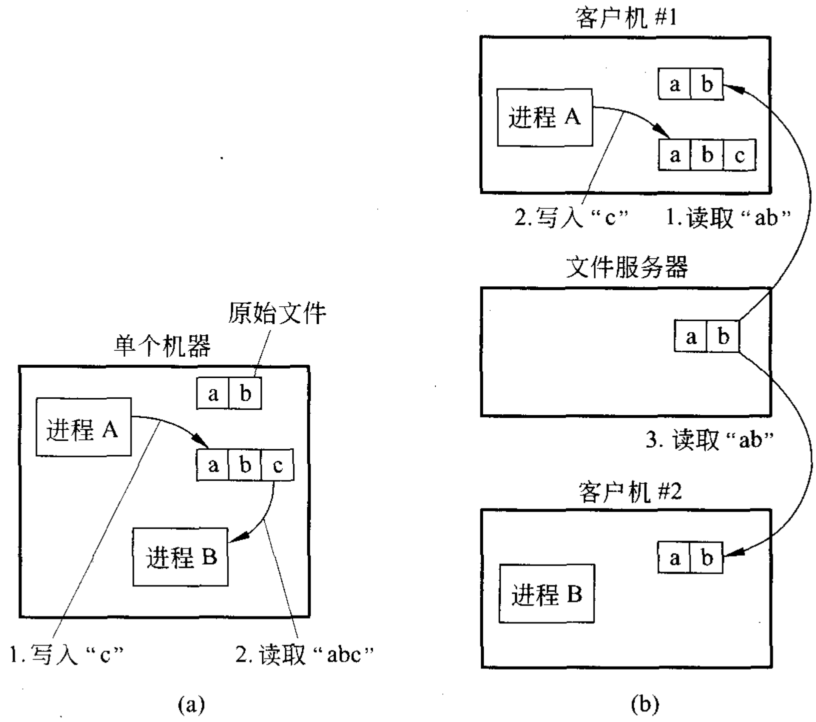


图 11.16

(a) 在单处理器上，当 read 操作出现在 write 操作之后时，read 操作返回的值是刚刚写入的值；
(b) 使用缓存的分布式系统可能返回过时的值

地修改缓存的文件，并且稍后另一个客户从该服务器读取这个文件，那么第二个客户会得到过时的文件，如图 11.16(b)所示。

解决这一难题的一种方法是立即将缓存文件的所有改动传播回服务器。尽管这种方法在概念上相当简单，但是它效率低。一种替代解决方案是放宽文件共享的语义。我们可以不要求 read 操作看到所有先前的 write 操作的结果，而定义一套新的规则：“对打开的文件所做的改动最初只对修改这个文件的进程(或者可能是机器)可见。只有当文件被关闭时，这些改动才对其他进程(或机器)可见。”采用这一规则不会改变图 11.16(b)所发生的情况，但是它确实将实际行为(B 得到文件的原始值)重新定义为正确的行为。当 A 关闭文件时，它向服务器发送一份副本以使随后的读操作像我们要求的那样得到新值。

这一规则被广泛地实现，并被称为**会话语义**(session semantics)。大多数分布式系统都实现了会话语义。这意味着：尽管它们在理论上遵循图 11.1(a)中的远程访问模型，但是大多数实现都使用本地缓存，从而有效地实现了图 11.1(b)中的上载/下载模型。

使用会话语义提出了以下问题：如果两个或更多的客户同时缓存和修改同一文件会发生什么情况。一种方法是在依次关闭每个文件时，将文件的值传递给服务器，所以最终的结果依赖于服务器最后处理的那个关闭请求。另一种稍差但相对易于实现的方法是：最终的结果是几个候选结果之一，但是并不指定选择哪个候选结果。

解决分布式系统中文件共享的语义的一种完全不同的方法是使所有文件都不可改变。因而，系统无法为写操作打开文件。结果，文件上的操作仅有 create 和 read。

创建一个全新的文件并使它以原先存在的文件名存入目录系统中是可能的，而此时那个原先存在的文件将变成不可访问的(至少不能以那个名称访问)。因而，尽管修改文件 x

变成不可能的，但是用一个新文件自动替换 x 仍然是可能的。也就是说，尽管文件不能更新，但是目录能更新。一旦我们决定文件根本不能改变，如何处理两个进程——其中一个进程写文件，而另一个进程读同一文件——的问题也就随之消失了。

剩下的问题是两个进程试图同时替换同一文件时会发生什么情况。就像会话语义一样，这里的最佳解决方案似乎是允许使用其中一个新文件替换旧文件，这可以是使用最后一个新文件或者不确定使用哪个新文件。

一个更复杂一些的问题是：如果一个文件被替换了，而另一个进程正忙于读取该文件，那么如何处理这种情况。一种方法是设法安排读取文件的进程继续使用旧文件，即使该文件可能不再存在于任何目录中，这种方法类似于 UNIX 允许打开文件的进程继续使用该文件，即使已经从所有目录中删除了该文件时也是如此。另一种方法是检测文件已经被改变并使随后试图从该文件读取数据的请求失败。

第四种处理分布式系统中的共享文件的方法是使用原子/%ic 事务。简要地总结一下，为访问一个文件或一组文件，一个进程先执行某种类型的 BEGIN_TRANSACTION 原语以标志其后的语句必须不可分割地执行。然后是读、写一个或多个文件的系统调用。所有工作都完成后，执行一个 END_TRANSACTION 原语。这种方法的关键性质是：服务器保证包含在一个事务内的所有调用都会被按顺序执行，而不会受到其他并发事务的干扰。如果两个或更多的事务同时启动，那么系统保证最终的结果就像它们都按照某种（未定义的）顺序运行一样。

图 11.17 总结了已讨论的 4 种处理分布式系统中的共享文件的方法。

方法	注释
UNIX 语义	一个文件上的每个操作对所有进程都是即时可见的
会话语义	在文件关闭之前，所有改动对其他进程都是不可见的
不可改变的文件	不允许更新文件；简化了共享和复制
事务	所有改动都以原子方式发生

图 11.17　4 种处理分布式系统中的共享文件的方法

11.5.2　文件锁定

由于客户-服务器体系结构中具有无状态服务器，我们需要额外的机制来同步对共享文件的访问。执行该任务的传统方式是利用锁管理器。毫不例外，锁管理器遵循集中式锁定模式，如我们在第 6 章中所讨论的那样。

不过，事情并不像我们刚才描述的那样简单。尽管一般会部署锁管理器，由于需要允许并发地访问相同的文件，导致锁定机制变得复杂。为此开发了大量不同的锁，此外，锁的粒度也可能不同。让我们再次考虑 NFS 版本 4。

从概念上讲，NFS 版本 4 中的文件锁定比较简单。它实质上只提供 4 种与锁定相关的操作，如图 11.18 所示。NFS 对读操作锁和写操作锁加以区分。如果多个客户只读取数据，那么它们可以同时访问一个文件的相同部分。为了获得修改文件的某个部分的独占访问权，需要使用写操作锁。

操作	描述
lock	为一定范围的字节创建锁
lockt	测试是否授予了相冲突的锁
locku	删除一定范围的字节上的锁
renew	继续租用指定的锁

图 11.18　NFS 版本 4 中关于文件锁定的操作

lock 操作用于请求对文件的一段连续范围的字节加读操作锁或写操作锁。它是一个非阻塞操作，如果由于另一个相冲突的锁的存在而不能授予锁，那么客户会得到一条错误消息，并不得不在稍后某个时间轮询服务器。它不会自动进行重试。另一种可选择的方法是，客户可以请求将其放入服务器维护的 FIFO 列表中。一旦删除了相冲突的锁，服务器将把下一个锁授予位于该列表前端的客户，只要该客户在一定的时间期限内轮询服务器即可。这种方法避免了服务器必须通知客户的操作，与此同时对那些未能授予锁请求的客户仍然保持了公平，因为锁的授予是按照 FIFO 顺序进行的。

lockt 操作用于测试是否存在相冲突的锁。例如，客户在请求一个文件的指定范围的字节上的写操作锁之前，可以测试哪些字节是否已经被授予了读操作锁。在发生冲突的情况下，提出请求的客户被告知哪个客户造成这个冲突，以及在哪个范围的字节上发生冲突。这个操作的实现可以比 lock 的实现更高效，因为它不需要打开文件。

删除一个文件上的锁是通过 locku 操作完成的。

被授予的锁在一段指定时间（由服务器决定）内有效。换句话说，它们都具有关联的租约。除非客户更新授予其锁上的租约，否则服务器会自动删除它的锁。我们将看到，服务器端提供的其他资源也采用这种方法，它有助于故障后的恢复。客户使用 renew 操作来请求服务器更新其锁上的租约（实际上，其他资源也是这样）。

除了这些操作之外，还有一种锁定文件的隐含方法，称为**共享预约**（share reservation）。共享预约完全独立于锁定，它可用于在基于 Windows 的系统上实现 NFS。客户打开文件时，它指定所需的访问类型（即 READ、WRITE 或 BOTH），以及服务器应该拒绝的其他客户的访问类型（NONE、READ、WRITE 或 BOTH）。如果服务器不能满足客户的需求，那么该客户的 open 操作将会失败。图 11.19 确切地表示了一个新客户试图打开一个已被另一个客户成功打开的文件时所发生的情况。对于一个已经打开的文件，我们用两个不同的状态变量加以区分。访问状态表明当前客户目前如何访问该文件。拒绝状态表明新客户不允许进行哪些访问。

图 11.19(a)表示在给定一个文件的当前拒绝状态的情况下，客户试图请求以指定的访问类型打开此文件时所发生的事情。同样，图 11.19(b)表示试图打开一个当前正被另一个客户访问的文件而所请求的访问类型不被该客户允许的情况。

关于为共享文件提供同步机制，NFS 版本 4 决不是一个例外。事实上，迄今为止人们只接受任何简单的原语集，比如只锁定完整的文件，这反映出设计不佳。锁定模式中的复杂性主要来自于以下事实：要求进行细粒度的锁定，以允许对共享文件进行并发访问。目前已经采取了一些尝试，以便在降低复杂性的同时保持性能（Burns 等 2001），但是得到的结果仍然不那么令人满意。最终，我们可能出于可伸缩性而考虑完全重新设计我们的应用程序

当前文件拒绝状态

请求访问		NONE	READ	WRITE	BOTH
	READ	成功	失败	成功	失败
	WRITE	成功	成功	失败	失败
	BOTH	成功	失败	失败	失败

(a)

请求的文件拒绝状态

当前访问状态		NONE	READ	WRITE	BOTH
	READ	成功	失败	成功	失败
	WRITE	成功	成功	失败	失败
	BOTH	成功	失败	失败	失败

(b)

图 11.19 使用 NFS 共享预约实现 open 操作的结果

(a) 在当前拒绝状态下,客户请求共享访问时的情况;(b) 在当前文件访问状态下,客户请求拒绝状态时的情况

11.5.3 在 Coda 中共享文件

NFS 中的会话语义规定最后一个关闭文件的进程将把它所做的改动传播到服务器;任何更新都是并发进行的,但是早期的会话将会丢失。也可以采用一种更微妙一点的方法。为了提供文件共享,Coda 文件系统(Kistler 和 Satyanaryanan,1992)使用一种特殊的分配模式,该模式与 NFS 中的共享预约存在某些相似之处。下面的内容对于理解该模式的工作方式十分重要。当一个客户成功地打开一个文件 f 时,f 的完整副本都将被传递给客户机。服务器会记录客户具有 f 的副本。迄今为止,这种方法类似于 NFS 中的开放式委托。

现在假设客户 A 已为写操作打开文件 f。当另一个客户 B 也想打开 f 时,它会失败。这是由于服务器已经具有客户 A 可能已经修改 f 的记录引起的。另一方面,如果客户 A 为读操作打开 f,那么客户 B 试图从服务器获得一份副本以执行读操作将会成功。B 试图为写操作打开文件也将会成功。

现在考虑一下 f 的多个副本已经存储于多个客户本地的情况。假定像我们刚才所说的那样,只有一个客户能够修改 f。如果这个客户修改 f,随后关闭该文件,那么文件将被传回服务器。不过,其他各个客户可能继续读取它们的本地副本,尽管实际上它们的副本是过时的。

导致这种明显不一致行为的原因在于 Coda 将会话视为事务处理。以图 11.20 为例,它显示了两个进程 A 和 B 的时间线。假设 A 已为读操作打开 f,它导致会话 S_A。客户 B 已为写操作打开 f,被显示为会话 S_B。

当 B 关闭会话 S_B 时,它向服务器传递 f 的更新版本,然后服务器将向 A 发送一个无效消息。此时,A 知道它正在读取较老版本的 f。但是,从事务的角度来说,这真的无关紧要,因为可以认为会话 S_A 已经安排在会话 S_B 之前。

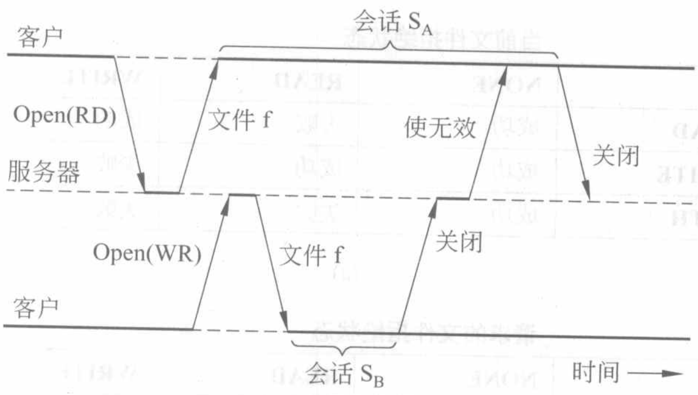


图 11.20　Coda 中共享文件时的事务行为

11.6　一致性和复制

缓存和复制在分布式文件系统中扮演着重要角色，最引人注目的是它们被设计成在广域网上工作。接下来，我们将看看与文件数据的客户端缓存相关的多个不同的方面，以及文件服务器的复制。此外，我们还将考虑对等的文件共享系统中的复制的作用。

11.6.1　客户端缓存

为了查看在实际中如何部署客户端缓存，我们回到我们的示例系统 NFS 和 Coda 上来。

11.6.1.1　NFS 中的缓存

NFS 版本 3 中的缓存主要在协议外部处理。这种方法导致了不同缓存策略的实现，而其中大部分实现从不保证一致性。与存储在服务器上的数据相比，在最理想的情况下，缓存的数据可能在几秒钟之内就过时了。但是，允许缓存的数据在客户完全不知晓的情况下在 30 秒内过时的实现也是存在的。这种情况就很难令人满意。

NFS 版本 4 解决了一些有关一致性的问题，但它实质上仍以实现相关的方式处理缓存一致性。图 11.21 显示了 NFS 假定的通用缓存模型。每个客户都可以有一个存储器缓存，它包含先前从服务器读取的数据。另外，客户端还有一个磁盘缓存，它作为存储器缓存的扩展，并使用相同的一致性参数。

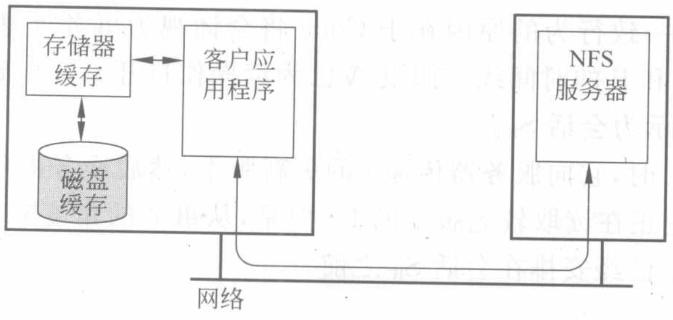


图 11.21　NFS 中的客户端缓存

通常，客户缓存文件数据、属性、文件句柄和目录。可采用不同的策略处理缓存数据、缓存属性等的一致性。下面，我们先看看缓存文件数据的情况。

NFS 版本 4 支持两种缓存文件数据的方法。最简单的方法是当客户打开文件时，将它从服务器获得的数据缓存起来，并把这些数据作为各种 read 操作的结果。另外，write 操作也可以在缓存中执行。客户关闭文件时，NFS 要求如果文件被修改，那么必须把缓存的数据送回服务器。这种方法对应于实现以前讨论的会话语义。

一旦文件(的一部分)被缓存，客户就可以在缓存中保留它的数据，即使在关闭文件之后亦可如此。同一台机器上的多个客户也可以共享一个缓存。NFS 要求：每当客户打开一个以前被关闭但(部分)被缓存的文件，客户必须马上使缓存的数据重新有效。这是通过检查文件最后被修改的时间并使包含过时数据的缓存无效来实现的。

在 NFS 版本 4 中，在打开文件时，服务器可以将它的某些权限委托给客户。**开放式委托**(open delegation)发生在客户机被允许在本地处理来自同一机器的其他客户的 open 和 close 操作时。例如，因为需要考虑共享预约，所以常常由服务器负责检查打开文件的操作是否成功。利用开放式委托，有时允许客户机执行这种检查，以避免联系服务器的需要。

例如，如果服务器将文件的打开操作委托给请求写访问权限的客户，那么来自同一台机器上的其他客户的文件锁定也可以在本地处理。服务器仍会处理来自其他机器上的客户的锁定请求，它只会拒绝那些客户访问文件。注意，将文件委托给请求只读访问权限的客户时，这种模式无法工作。在这种情况下，每当另一个本地客户想要获得写访问权限时，该客户将不得不联系服务器，在本地处理这个请求是不可能的。

将文件委托给客户的一个重要后果是：服务器需要有能力撤销这种委托，例如，当另一个位于不同机器上的客户需要获得该文件的访问权时，服务器就需要撤销文件的委托。撤销委托要求服务器能够回调客户，如图 11.22 所示。

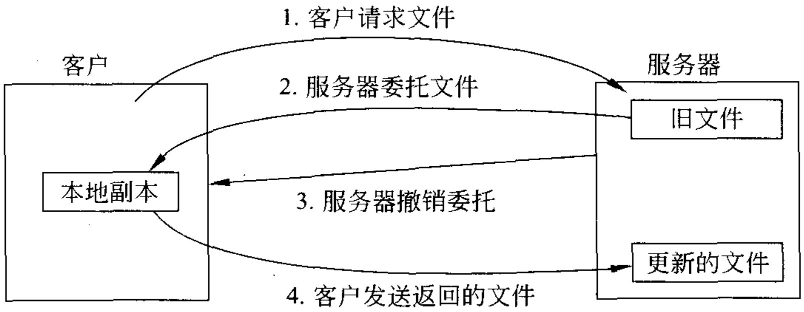


图 11.22 使用 NFS 版本 4 的回调机制撤销文件委托

在 NFS 中，回调是通过 NFS 底层的 RPC 机制实现的。不过，注意：回调要求服务器记录已委托文件的客户。这里，我们再看一个示例，该示例中的 NFS 服务器不再以无状态的方式实现。不过，注意：将委托和有状态的服务器结合起来可能在出现客户和服务器故障时引发各种问题。例如，当服务器把一个文件委托给现在无反应的客户时服务器应该做什么？我们稍后将会讨论，租约一般会构成充分实用的解决方案。

客户也可以缓存属性值，但是客户何时能够使缓存的属性值保持一致主要取决于客户自身。特别是，两个不同客户缓存的同一文件的属性值可能有所不同，除非这两个客户共同使这些属性保持一致。对属性值的修改应该立即转发给服务器，这样就遵循了直写式缓存

相关性策略。

缓存文件句柄(更准确地讲是名称-文件句柄映射)和目录也使用相似的方法。NFS 使用缓存的属性、文件句柄和目录上的租约来减轻不一致性的影响。由此，一段时间后，客户自动使缓存数据项无效，再次使用这些数据项之前，需要使它们重新变得有效。

11.6.1.2 Coda 中的客户端缓存

客户端缓存对 Coda 的操作至关重要，这是由于以下两个原因。第一，进行缓存是为了实现可伸缩性。第二，缓存提供了较高程度的容错性，这是因为客户变得较少地依赖于服务器的可用性。出于这两个原因，Coda 中的客户总会缓存整个文件。也就是说，不管文件是为读操作还是为写操作而打开的，都会把文件的整个副本传送给客户，然后客户将会缓存这份副本。

与许多其他的分布式文件系统不同，Coda 中的缓存相关性是通过回调维护的。我们在讨论文件共享语义时已经遇到过这种现象。对于每个文件，客户从服务器获取文件时，服务器将会记录哪些客户在本地缓存了该文件的副本。此时称服务器为客户记录**回调承诺**(callback promise)。当客户第一次更新该文件的本地副本时，它会通知服务器，然后，服务器依次向其他客户发送无效消息。这种无效消息称为**回调中断**(callback break)，这是因为服务器随后将会废弃它为刚刚向其发送了无效消息的客户保存的回调承诺。

这种模式的有趣方面在于：只要客户知道它在服务器上有未被废弃的回调承诺，它就可以安全地在本地访问文件。特别是，假设客户打开一个文件，它发现该文件已经在它的缓存中。这样的话，如果服务器仍为这个客户保留着该文件的回调承诺，那么这个客户就可以使用该文件。客户必须与服务器核对那个回调承诺是否还存在。如果是，服务器就不需要再向客户传送文件了。

这种方法如图 11.23 所示，它是图 11.20 的扩展。当客户 A 开始会话 S_A 时，服务器会记录一个回调承诺。同样，B 开始会话 S_B 时也是如此。不过，当 B 关闭 S_B 时，服务器向客户 A 发送回调中断，从而中断了它对回调客户 A 的承诺。注意：由于 Coda 的事务处理语义，当客户 A 关闭会话 S_A 时，不会发生任何特殊的事情，这个关闭操作就像所预期的那样被简单地接受。

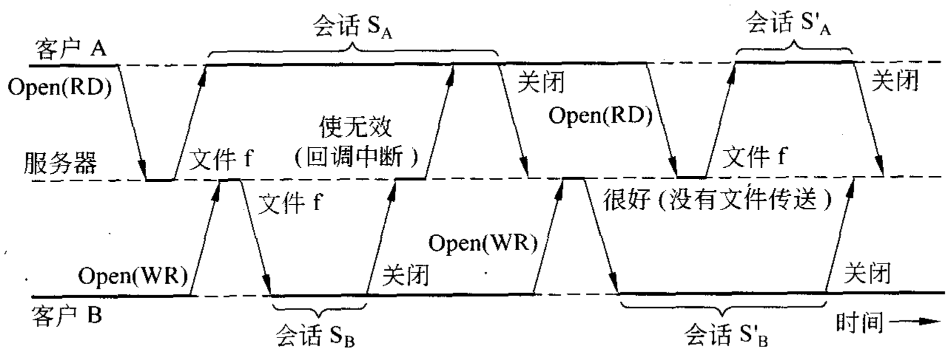


图 11.23　在 Coda 中打开会话时本地副本的使用

因此，当 A 随后要打开会话 S'_A 时，它会发现其 f 本地的副本是无效的，所以它必须从服务器获取最新的版本。另一方面，当 B 开始会话 S'_B 时，它会注意到服务器仍有一个未被废

弃的回调承诺，该承诺意味着B可简单地重用从会话 S_B 获得的本地副本。

11.6.1.3 用于便携式设备的客户端缓存

许多分布式系统的一个重要的发展是：不再能够假定许多存储设备将通过网络永久地连接到系统上。相反，用户具有多种不同类型的半永久性连接的存储设备，例如，通过发射台或者对接站(docking station)进行连接。典型的示例包括PDA、笔记本计算机设备，还包括便携式多媒体设备，比如电影和音频播放器。

在大多数情况下，将使用一种显式的上载/下载模型在便携式存储设备上维护文件。如果存储设备被视作分布式文件系统的一部分，就可以使事情变得简单。在这种情况下，无论何时需要访问一个文件，都有可能从本地设备或者通过到达系统其余部分的连接获取它。需要区分这两种情况。

(Tolia等 2004)提议采用一种非常简单的方法，即在本地存储文件中所包含数据的加密散列。这些散列存储在便携式设备上，并且用于重定向对关联内容的请求。例如，当在本地存储目录列表时，将不会存储每个列出的文件的数据，而只会存储计算的散列。这样，当获取一个文件时，系统将首先检查文件是否在本地可用并且是否是最新的。注意：过时的文件所具有的散列将不同于目录列表中存储的散列。如果文件在本地可用，就可以把它返回给客户，否则将需要进行数据传输。

显然，当断开一个设备时，它将不可能传输任何数据。可以利用多种技术来确保高度的便携性，即尽可能地把将要使用的文件确实存储在设备本地。与大多数缓存模式所固有的按需进行数据传输的方法相比，在这些情况下我们需要部署预先获取文件的技术。不过，对于许多便携式存储设备，我们可以期望用户将使用特殊的程序在设备上预先安装一些文件。

11.6.2 服务器端复制

与客户端缓存相比，分布式文件系统中的服务器端复制不那么常见。当然，当可用性至关重要时，就会应用复制，但是从性能的角度讲，部署缓存使得其中的整个文件或者文件的较多部分可供客户本地使用将显得更有意义。为什么客户端缓存如此受欢迎的一个重要原因是：实践证明文件共享相对比较少见。当共享发生时，通常只用于读取数据，在这种情况下缓存是一种优秀的解决方案。

对于性能来讲，服务器端复制的另一个问题是：高度的复制和较低的读/写比率的组合实际上可能使性能降级。在认识到需要在每个副本上执行每个更新操作时，就很容易理解这一点。换句话说，对于一个N倍的复制文件，单个更新请求将导致更新操作增加N倍。此外，需要对并发更新进行同步，从而导致更多的通信，并且会进一步降低性能。

出于这些原因，一般仅为容错性考虑时才会复制文件服务器。下面，我们将说明Coda文件系统中的这种类型的复制。

Coda中的服务器复制

Coda允许复制文件服务器。如前所述，复制的单位是称为卷(volume)的文件集合。实质上讲，卷对应于UNIX磁盘分区，也就是像这样直接受操作系统支持的传统文件系统，尽

管卷一般要小得多。拥有一份卷副本的 Coda 服务器的集合称为该卷的**卷存储组**(volume storage group)，或简称为 **VSG**。出现故障时，客户可能不能访问卷的 VSG 中的所有服务器。对于一个卷，客户的**可访问卷存储组**(accessible volume storage group，AVSG)是由卷的 VSG 中客户此刻可以联系的那些服务器组成的。如果 AVSG 是空的，那么称该客户是**断开**(disconnected)的。

Coda 使用复制的写协议来维护复制卷的一致性。特别是，它使用第 7 章介绍的 ROWA(Read-One, Write-All)的一种变体。当客户需要读取一个文件时，该客户联系该文件所属卷的 AVSG 中的一个成员。不过，在关闭一个更新文件上的会话时，客户将把这个更新文件并行地传送给 AVSG 中的每个成员。这种并行传送是通过如前所述的 MultiRPC 实现的。

只要没有故障，即在对于每个客户来说客户卷的 AVSG 与该卷的 VSG 相同的情况下，这种模式工作得很好。但是，在出现故障时，情况可能变得很糟。考虑一个跨三个服务器 S_1、S_2 和 S_3 复制的卷。对于客户 A，假设它的 AVSG 包括服务器 S_1 和 S_2，而客户 B 只有权访问服务器 S_3，如图 11.24 所示。

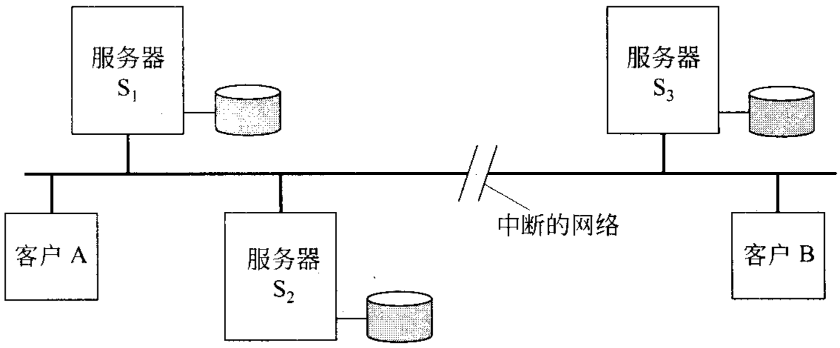


图 11.24　两个客户具有用于同一复制文件的不同 AVSG

Coda 使用一种优化策略实现文件复制。具体地说，A 和 B 都被允许为执行写操作打开文件 f，更新它们各自的副本，并将它们的副本传回它们的 AVSG 中的成员。显然，在 VSG 中将存在 f 的不同版本。问题是如何检测和解决这种不一致性。

Coda 采用的解决方案是部署一种版本记录模式。具体地说，VSG 中的服务器 S_i 为该 VSG 包含的每个文件 f 维护一个 Coda 版本向量(coda version vector) $CVV_i(f)$。如果 $CVV_i(f)[j] = k$，那么服务器 S_i 就知道服务器 S_j 至少已经看到文件 f 的版本 k。$CVV_i(f)[i]$是存储在服务器 S_i 上的 f 的当前版本号。服务器 S_i 对 f 的更新会导致 $CVV_i(f)[i]$增加。注意，版本向量完全类似于第 6 章讨论的向量时间戳。

回到刚才讨论的三个服务器的示例上来，对于每个服务器 S_i，$CVV_i(f)$的初值是[1,1,1]。当客户 A 从其 AVSG 中的一个服务器(比如说 S_1)读取 f 时，它也会收到 $CVV_1(f)$。更新 f 后，客户 A 向其 AVSG 中的每个服务器(即 S_1 和 S_2)多播 f。然后，这两个服务器都会记录它们各自的副本已被更新，但是 S_3 的副本没有被更新。也就是说：

$$CVV_1(f) = CVV_2(f) = [2,2,1]$$

此时，客户 B 被允许打开一个会话，在该会话中，它从服务器 S_3 接收一份 f 的副本，并随后更新 f。当它关闭这个会话并向 S_3 传递更新时，服务器 S_3 会把它的版本向量更新为 $CVV_3(f) = [1,1,2]$。

当分区结束时，这三个服务器需要使用它们的副本重新合成 f。通过比较它们的版本向量，它们就会发现出现了需要进行修复的冲突。在很多情况下，可以使用应用程序相关的方法自动地解决冲突，有关内容在(Kumar 和 Satyanarayanan 1995)中有所讨论。不过，在很多情况下，用户不得不参与手工解决冲突，特别是当不同用户以不同的方式修改了同一文件的相同部分时。

11.6.3　对等文件系统中的复制

让我们现在研究对等文件共享系统中的复制。复制在这里也扮演着重要角色，特别是加快搜索和查找请求的速度，还可以平衡结点之间的负载。这些系统中的一个重要性质是几乎所有的文件都是只读的。更新只包括向系统中添加文件这种形式。应该区分非结构化和结构化对等系统。

11.6.3.1　非结构化对等系统

非结构化对等系统的基本原理是：将查找数据归结为搜索网络中的数据。实际上，这意味着(例如)一个结点只需将搜索查询广播给它的邻居，然后这些邻居可能会转发查询，依此类推。显然，通过广播方式进行搜索一般不是一个好主意，需要采取特殊的措施来避免性能问题。(Risson 和 Moors 2006)中详细讨论了在对等系统中执行搜索。

与广播受到限制的方式无关，显然，如果文件被复制，那么搜索将变得更容易、更快速。一种极端情况是在所有结点上复制一个文件，这意味着完全可以在本地搜索任何文件。不过，如果结点具有有限的容量，完全复制将是不可能的。这样，问题就变成寻找一种最优的复制策略，这里的最优性是由在找到文件前需要处理特定查询的不同结点数定义的。

(Cohen 和 Shenker 2002)研究了这个问题，他们假定文件复制是可以控制的。换句话说，假定可以指示非结构化对等系统中的结点保存文件的副本，那么把文件副本分配给结点的最佳策略是什么呢?

让我们考虑两种极端情况。一种策略是跨整个网络统一分配每个文件的 n 个副本。这种策略忽略了不同的文件可能具有不同的请求比率，也就是说，有些文件可能比其他文件更受欢迎。另一种替代策略是根据文件被搜索的频率来复制文件：文件越受欢迎，我们就创建和分配越多的副本以覆盖整个网络。

顺便说一下，注意：后一种策略可能使得分配不受欢迎的文件的代价非常高昂。可能令人奇怪的是，从经济的角度讲，这种搜索可能被证明越来越重要。推理很简单：随着因特网允许人们快速、容易地访问大量的信息，开拓缝隙市场(niche market)突然变得非常吸引人。因此，如果你有兴趣获取适当的设备(比如一辆斜靠式自行车)，因特网就提供了这种搜索能力，它允许你高效地发现合适的销售商。

十分令人奇怪的是，事实证明：在考虑需要被查询的平均结点数时，统一的、受欢迎的策略执行得一样很好。在两种情况下，查询的分布是相同的，因此在受欢迎的策略中文档的分布遵循查询的分布。此外，事实证明：这两种策略“之间”的任何分配都更好一些。获得这样一种分配是切实可行的，而不是毫无价值的。

当用户从其他人那里下载文件然后使它们公共可用时，非结构化对等系统中的复制就

会自然地发生。除了由单个组织控制某些部分之外，在实际中很难控制这些网络。此外，BitTorrent 实施的研究表明：在涉及复制文件并使它们可用时，还存在一种重要的社会因素（Pouwelse 等 2005）。例如，一些人展示了无私的行为，或者在他们完成其下载后使文件不再可用。问题是：是否能够设计出利用这种行为的系统。

11.6.3.2 结构化对等系统

考虑结构化对等系统中的查找操作的效率，复制主要被部署用于平衡结点之间的负载。我们在第5章中已经遇到过复制的"结构化"形式，（Ramasubramanian 和 Sirer 2004b）利用的复制形式甚至能够把平均查找步骤减至 O(1)。不过，在谈到负载平衡时，需要探讨不同的方法。

一种常见的实用方法是：沿着查询从源到目的地所经过的路径简单地复制一个文件。这种复制策略的效果是：将把大多数副本存放在负责存储文件的结点附近，因此当请求发生的频率很高时，这确实可以减轻结点的负载。不过，这种复制策略不会把其他结点的负载考虑在内，因此可能比较容易导致不平衡的系统。

为了解决这些问题，（Gopalakrishnan 等 2004）提出了一种不同的模式，它把沿着查询路径分布的结点上的当前负载考虑在内。其主要思想是：在查询的源结点上存储副本，并在沿着从源到目的地的查询路径上的结点中缓存指向这些副本的指针。更确切地讲，当从结点 P 到 Q 的查询经过结点 R 时，R 将检查是否应该把它的任何文件卸载给 P。它通过简单地查看它自己的查询负载来执行该任务。与施加到 P 上的负载相比，如果 R 要服务的对它所存储文件的查询请求过多，它可以要求 P 安装 R 的大多数被请求文件的副本。图 11.25 中描述了这种原理。

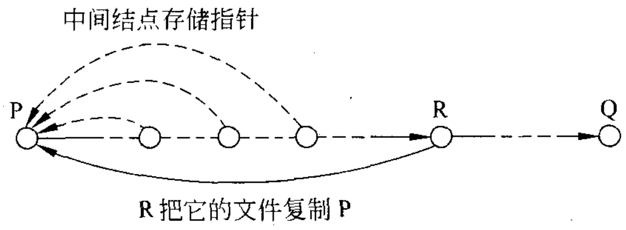


图 11.25 在对等系统中通过复制平衡负载

如果 P 可以接受来自 R 的文件 f，在从 P 到 R 的路径上访问的每个结点都将把用于 f 的指针安装到 P 上，指示可以在 P 上找到 f 的副本。

显然，传播副本存储在什么位置的信息对于让这种模式正确工作是非常重要的。因此，当通过覆盖技术路由查询时，结点可能还会传递关于它宿主的副本的信息。这种信息可能导致进一步安装指针，从面允许结点正确地决定把请求重定向到保存所请求文件的副本的结点。这些指针存放在一个大小有限的缓存中，并遵照最近最少使用策略替换它们（也就是说，如果缓存的指针所引用的文件从未被请求过，则将会迅速删除这些指针）。

11.6.4 网格系统中的文件复制

作为我们关于文件复制的最后一个主题，让我们考虑在网格计算中会发生什么事情。自然，性能在这个领域扮演着至关重要的角色，因为许多网格应用都是高度计算密集型的。

此外，我们看到一些应用通常还需要处理大量的数据。因此，在将要执行应用的地方，人们在复制文件方面投入了大量的工作。不过，执行该任务的方式简单得(有点)令人吃惊。

人们观察到的一个关键的事实是：在许多网格应用中数据是只读的。数据通常是从传感器或其他应用产生的，但是在产生并存储它们之后，极少对它们进行更新或修改。因此，数据复制可能广泛发生，事实上的确如此。

不幸的是，数据集的规模有时是如此之大，以至于需要采取特殊的措施来避免数据提供者(即那些存储数据集的机器)由于它们需要通过网络传送的数据量而变得过载。另一方面，由于经常要复制大量的数据，平衡获取副本的负载就是一个次要问题了。

网格系统中的复制主要是关于定位复制数据的最佳来源的问题。可以通过特殊的**副本定位服务**(replica location service)来解决这个问题，它非常类似于我们为命名服务讨论的定位服务。为 Globus 工具包开发的一种显而易见的方法是使用一种基于 DHT 的系统，比如用于分散式查找副本的 Chrod(Cai 等 2004)。在这种情况下，客户把一个文件名传递给服务的任何结点，在其中把它转换成一个键，随后查找它。返回到客户的信息包含所请求文件的联系地址。

为了使事情保持简单，随后使用类似于 FTP 的协议从多个站点下载定位的文件，然后客户可以利用复制定位服务注册它自己的副本。(Chervenak 等 2005)更详细地描述了这种体系结构，但是其方法相当简单。

11.7 容错性

分布式文件系统中的容错性是根据我们在第 8 章中讨论的原理处理的。如我们已经提及的，在许多情况下，将部署复制来创建容错服务器组。因此，在本节中，我们将重点介绍分布式文件系统的容错性方面的一些特殊问题。

11.7.1 处理 Byzantine 故障

在处理容错性时，经常被忽略的问题之一是：服务器可能展示任意的故障。换句话说，大多数系统不会考虑我们在第 8 章中讨论过的 Byzantine 故障。除了复杂性之外，忽略这类故障的原因是需要对执行环境做出有说服力的假设。特别是，必须假定通信延迟是有限制的。

在实际设置中，这种假设并不现实。鉴于此，(Castro 和 Liskov 2002)设计了一种可用于处理 Byzantine 故障的解决方案，它也可以在像 Internet 这样的网络中工作。我们将在这里讨论这种协议，因为它可以(并且已经)直接应用于分布式文件系统，特别是基于 NFS 的系统。当然，还有其他一些应用。其基本思想是：通过构造有限状态机的集合来部署主动复制，并且这个集合中具有无故障的进程以相同的顺序执行操作。假定至多有 k 个进程同时失败，客户发送一个操作给整个组，并接收至少由 k+1 个不同的进程返回的应答。

为了防止 Byzantine 故障，服务器组必须包含至少 3k+1 个进程。实现这种保护的难点在于：确保无故障的进程以相同的顺序执行所有的操作。达到这个目标的简单方式是：

指定一个协调器，它通过简单地给每个请求附加一个序号来序列化所有的操作。当然，问题是协调器也有可能失败。

这样，问题就转到了失败的协调器身上。与虚拟同步非常相似，进程会经过一系列视图，每个视图中的成员都会对无故障的线程达成一致意见，并且在当前主机看起来好像失败时启动视图更改。如果我们假定序号是一个接一个地分发的，就可以检测到主机失败，使得操作的间隙或者超时时间可以指示出现了某种错误。注意：进程可能错误地认为需要安装新的视图。不过，这不会影响系统的正确性。

该协议的一个重要部分依赖于如下事实：可以正确地对请求排序。最终，将使用一种法定机制：无论何时一个进程接收到一个请求，利用视图 v 中的数字 n 执行操作 o，它都会把这个请求发送给所有其他进程，并且等待，直到它从至少其他 2k 个进程接收到一条确认，指示它们已经看到了相同的请求。这样，我们就得到了请求的法定数量即 2k+1。这样一个确认被称为**法定证书**(quorum certificate)。实质上讲，它告诉我们有足够多的进程存储了相同的请求，因此它可以安全地继续执行操作。

整个协议包含 5 个阶段，如图 11.26 所示。

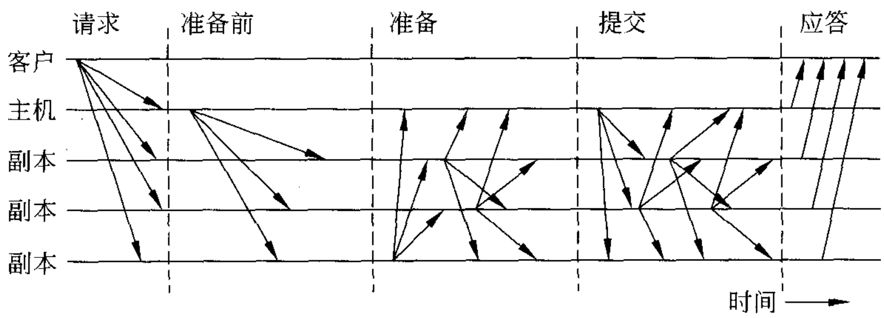


图 11.26 Byzantine 容错性中的不同阶段

在第一个阶段中，客户把一个请求发送给整个服务器组。一旦主机接收到请求，它就会在**准备前阶段**(pre-prepare phase)中多播一个序号，使得可以正确地对关联的操作进行排序。此时，从副本需要确保主机的序号会被法定成员接受，假定每个成员都会接受主机的提议。因此，如果从副本接受提议的序号，它就会把接受的序号多播给其他从副本。在**提交**(commit)阶段，将会达成协议，所有的进程都会相互通知并执行操作，然后客户最终可以看到结果。

当考虑多个阶段时，看起来似乎在**准备**(prepare)阶段之后所有进程都应该对请求的相同次序达成一致意见。不过，仅在同一个视图内是这样：如果需要改变为新的视图，不同的进程可能具有用于不同操作的相同序号，但它们是在不同视图中分配的。鉴于此，我们还需要**提交**(commit)阶段，在该阶段中，每个进程现在告诉其他进程它已经在其本地日志中存储了请求，并且针对的是当前视图。因此，即使需要从崩溃中恢复，一个进程也将会准确知道分配了哪些序号，以及是在哪个视图中分配的。

同样，一旦无故障的进程看到了相同的 2k 个提交消息（并且它们应该与自己的意图匹配），就可以执行提交阶段。同样，我们现在具有 2k+1 个法定进程用于执行操作。当然，应该先执行具有较低序号的挂起操作。

改变为新视图实质上遵循第 8 章中讨论过的用于虚拟同步的视图改变。在这种情况

下，进程需要发送关于它所知道的准备前消息的信息，以及从前一个视图中接收到的准备消息。我们在此将不进一步讨论详细的信息。

目前已经为基于 NFS 的文件系统实现了该协议，还实现了多种重要的优化和精心设计的数据结构，可以在(Castro 和 Liskov 2002)中找到关于它们的详细信息。有一种包装器允许把 Byzantine 容错性与遗留的应用结合起来，可以在(Castro 等 2003)中找到关于它的说明。

11.7.2　对等系统中的高度可用性

人们特别关注的一个问题是：确保对等系统中的可用性。一方面，似乎通过简单地复制文件即可轻松地保证可用性。不过，问题是结点的不可用性是如此之高，以至于这种简单的推理不再正确。如我们在第 8 章中所解释的，高度可用性的关键解决方案是冗余性。当涉及文件时，实质上有两种不同的方法来实现冗余性：复制和擦除编码。

擦除编码(erasure coding)是一种著名的技术，通过它把一个文件分成 m 块，随后把它们记录到 n>m 块中。这种编码模式的关键问题是：任何 m 个编码块的集合都足以用于重新构造原始文件。在这种情况下，冗余性因子等于 $r_{ec}=n/m$。假定平均结点可用性为 a，必需的文件不可用性为 ε，我们需要保证至少有 m 块可用，即：

$$1-\varepsilon\sum_{i=m}^{n}\begin{bmatrix}n\\i\end{bmatrix}a^{i}(1-a)^{n-i}$$

如果我们比较这种方法与复制文件方法，就会看到文件不可用性完全是由它所有的 r_{rep} 副本不可用的概率决定的。如果我们假定结点偏差是独立的并且具有完全相同的分布，就有：

$$1-\varepsilon=1-(1-a)^{r_{rep}}$$

应用一些代数运算和近似，我们可以通过考虑比率 r_{rep}/r_{ec} 与结点可用性 a 之间的关系，表达复制与擦除编码这两种方法之间的区别。这种关系如图 11.27 所示，我们为它设置 m=5[另请参见(Bhagwan 等 2004)以及(Rodrigues 和 Liskov 2005)]。

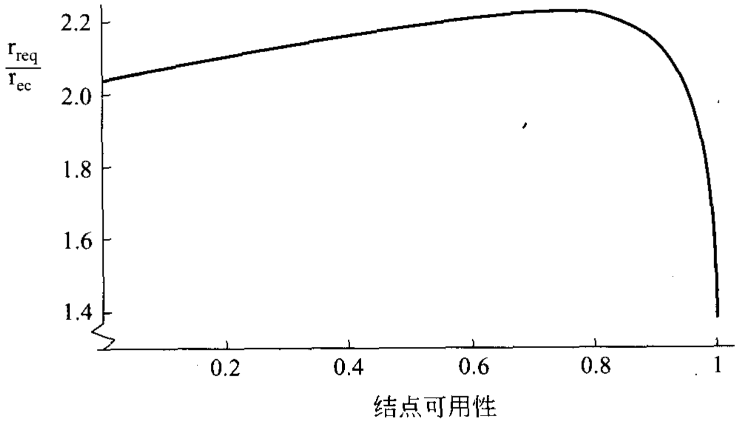


图 11.27　将比率 r_{rep}/r_{ec} 作为结点可用性 a 的函数

从这幅图中可以看出：在所有情况下，擦除编码所需的冗余性都比简单复制文件更少

一些。换句话说，从存储器角度讲，在其中的结点有规律地来来去去的对等网络中，复制文件以提高可用性比使用擦除编码技术的效率要低一些。

人们可能争辩说存储器中的这些节约实际上不再是一个问题，因为磁盘容量通常非常大。不过，实现这种维持冗余性的方法将会强制进行通信，这样较低的冗余性将节省带宽的使用。当结点通过非对称 DSL 或电缆线路（其中外出链路通常只有几百 Kbps 的容量）来响应连接到因特网的用户机器时，这种性能收益特别重要。

11.8 安 全 性

我们在第 9 章中讨论过的许多安全原理都直接适用于分布式文件系统。沿着客户-服务器体系结构组织的分布式文件系统中的安全性是让服务器处理身份认证和访问控制。这是一种处理安全性的直观方式，例如，在像 NFS 这样的系统中就采用了这种方法。

在这种情况下，通常具有单独的身份认证服务，比如 Kerberos，而文件服务器只处理授权。这种模式的主要缺点是：它需要集中管理用户，这可能会严重地妨碍可伸缩性。在下面的内容中，我们首先简要讨论 NFS 中的安全性，将其作为传统方法的一个例子，然后我们将重点关注一些替代方法。

11.8.1 NFS 中的安全性

如前所述，NFS 背后的基本思想是：远程文件系统应该像是客户的本地文件系统那样提供给客户。从这个角度讲，NFS 中的安全性主要集中于客户和服务器之间的通信就不足为奇了。安全通信意味着两者之间的安全信道应当以我们在第 9 章所讨论的方式建立。

除了安全 RPC 之外，控制文件访问也是十分必要的，这是通过 NFS 中的访问控制文件属性处理的。文件服务器负责验证其客户的访问权限，我们将在下面解释其工作方法。结合了安全 RPC 的 NFS 安全性体系结构如图 11.28 所示。

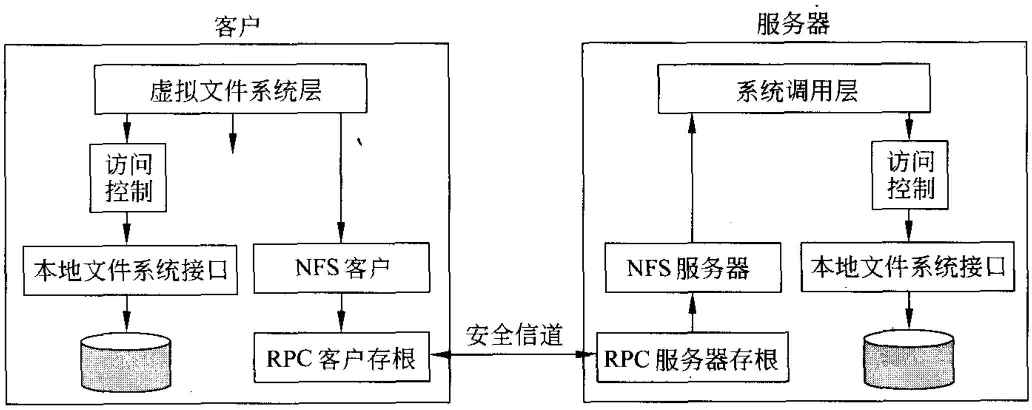


图 11.28 NFS 安全性体系结构

11.8.1.1　安全 RPC

因为 NFS 位于 RPC 系统的层次之上，所以建立 NFS 中的安全信道的关键在于建立安全 RPC。在 NFS 版本 4 之前，安全 RPC 仅意味着负责进行身份认证。进行身份认证可采用三种方法。现在，我们将依次研究这三种方法。

应用最广泛的方法称为系统身份认证，它实际上几乎不能进行任何身份认证。在这种基于 UNIX 的方法中，客户只向服务器发送它的有效用户 ID、组 ID 以及它声称自己所属于的组的列表。这一信息是以无签名的明文形式发送到服务器的。也就是说，服务器无法证实发送者声称的用户和组标识符是真正与该发送者关联的标识符。实质上讲，服务器假设客户已经通过一个正确的登录过程，并假设服务器可以信任该客户所在的机器。

老版本 NFS 所用的第二种身份认证方法是使用 Diffie-Hellman 密钥交换建立一个会话密钥，所以使用这种身份认证方法的 NFS 被称为安全 **NFS**(secure NFS)。我们在第 9 章中解释了 Diffie-Hellman 密钥交换的工作方式。这种方法比系统身份认证方法好很多，但是它比较复杂，也正是因为这一点，它较少得到实现。Diffie-Hellman 可以被看作是一个公钥加密系统。最初，它无法安全地分发服务器的公钥，但是由于后来引入了安全名称服务，使这一问题迎刃而解。长期以来，批评者一直批评它使用的是相对较小的公钥，NFS 中的公钥只有 192 位。事实证明，破解一个使用如此短的密钥的 Diffie-Hellman 系统几乎是毫不费力的(Lamacchia 和 Odiyzko 1991)。

第三种身份认证协议是 Kerberos，我们也曾在第 9 章中讨论过它。

随着 NFS 版本 4 的引入，由于它支持 RPCSEC_GSS，因此增强了安全性。**RPCSEC_GSS** 是一个通用的安全性框架，它可以支持各种建立安全信道的安全性协议(Eisler 等 1997)。特别是，它不仅支持不同身份认证系统的挂钩(hook)，而且支持消息的完整性和机密性，这两种特性是老版本 NFS 无法支持的。

RPCSEC_GSS 基于安全性服务的标准接口，该接口称为 GSS-API，(Linn 1997)中完整地描述了此接口。RPCSEC_GSS 所在的层在此接口所在的层之上，图 11.29 显示了其组织结构。

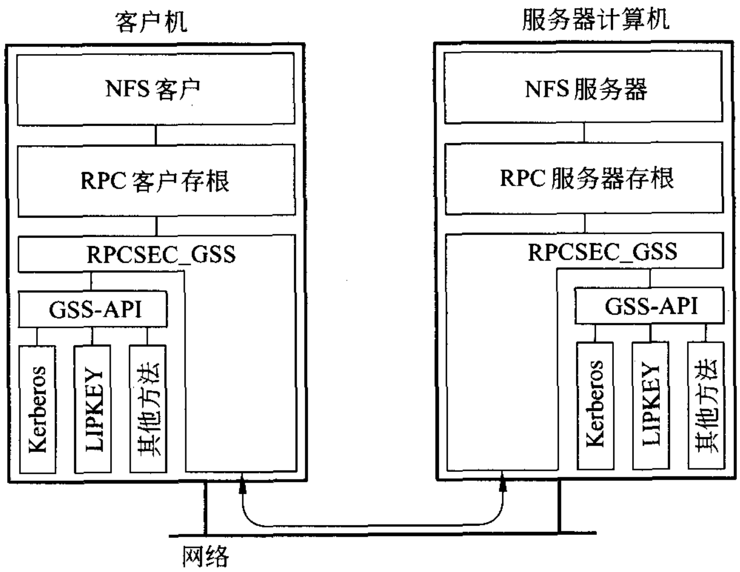


图 11.29　NFS 版本 4 中的安全 RPC

对于 NFS 版本 4，RPCSEC_GSS 应该配置为支持 Kerberos 版本 5。另外，系统也必须支持一个称为 **LIPKEY** 的方法，该方法在（Eisler 2000）中有所描述。LIPKEY 是一种公钥系统，它允许使用密码验证客户，而使用公钥验证服务器。

NFS 的安全 RPC 的重要部分是设计者可以选择不仅支持他们自己的安全性机制，而且支持处理安全性的标准方法。因此，已被证明的安全性机制（如 Kerberos）可以结合到 NFS 实现之中，而不会影响系统的其他部分。此外，如果现有的安全性机制被证明有缺陷（例如，使用小密钥的 Diffie-Hellman），那么替换它是十分容易的。

应该注意，因为 RPCSEC_GSS 是作为 RPC 层的一部分实现的，RPC 层位于 NFS 协议层之下，所以它也可以用于老版本的 NFS。不过，仅当引入 NFS 版本 4 之后，RPC 层的这种改变才变得可用。

11.8.1.2 访问控制

NFS 中的授权类似于安全 RPC：它提供一些机制，但是不指定任何特定的策略。访问控制是通过 ACL 文件属性实现的。这一属性是访问控制项目的列表，其中每个项目指定某个特定用户或组的访问权限。NFS 所区分的访问控制操作相对比较直观，包括那些用于读、写以及执行文件、操纵文件属性、列出目录等的访问控制。

值得注意的还是 synchronize 操作，该操作实质上指示与服务器共存的进程是否可以为绕过 NFS 协议直接访问文件，从而改进性能。NFS 访问控制模型具有比大多数 UNIX 模型丰富得多的语义。这种差异来自于 NFS 应能够与 Windows 2000 系统互操作的需求。其基本思想是使 UNIX 的访问控制模型更易于适应 Windows 的访问控制模型，反之亦然。

访问控制有别于诸如 UNIX 之类的文件系统的另一个方面是：可以为不同的用户和不同的组指定访问权限。传统上讲，对文件的访问是为单个用户（该文件的所有者）、单个用户组（例如，项目团队的成员）和所有其他人指定的。NFS 具有很多不同类型的用户和进程，如图 11.30 所示。

用户类型	描述
owner	文件的所有者
group	与文件关联的用户组
everyone	任何用户或进程
interactive	任何从交互式终端访问文件的进程
network	任何通过网络访问文件的进程
dialup	任何通过服务器的拨号连接访问文件的进程
batch	任何作为批处理作业的一部分访问文件的进程
anonymous	任何无需进行身份验证即可访问文件的用户
authenticated	任何经过身份证验证的用户或进程
service	任何系统定义的服务进程

图 11.30　NFS 所区分的关于访问控制的各种用户和进程

11.8.2 分散式身份认证

像 NFS 这样的系统的主要问题之一是：为了正确地处理身份认证，必须通过集中式系

统管理注册用户。通过结合使用**安全文件系统**(Secure File System,SFS)和分散式身份认证服务器,提供了这个问题的一种解决方案。其基本思想(Kaminsky 等 2003)十分简单。其他系统所欠缺的是:不能为一个用户指定远程用户在其文件上具有某些特权。在几乎所有的情况下,用户必须在全局为所有身份认证服务器所知。一种更简单的方法是:让 Alice 指定"可以在 X 上找到其详细信息的 Bob"具有某些特权。处理 Alice 的凭证的身份认证服务器然后就可以联系服务器 X,获取关于 Bob 的信息。

要解决的一个重要问题是:让 Alice 的服务器确切知道它正在与 Bob 的身份认证服务器打交道。可以使用自证明的名称解决这个问题,SFS 中引入了这个概念(Mazieres 等 1999),其目的是将密钥管理与文件系统安全性分隔开。SFS 的整体组织结构如图 11.31 所示。为了确保其在多种类型的机器上的可移植性,SFS 已与多种 NFS 版本 3 组件集成在一起。客户机上有三个不同的组件,其中不包括用户的程序。NFS 客户被用作用户程序的接口,它与 **SFS 客户**(SFS client)交换信息。对于 NFS 客户来说,SFS 客户就好像是另一个 NFS 服务器。

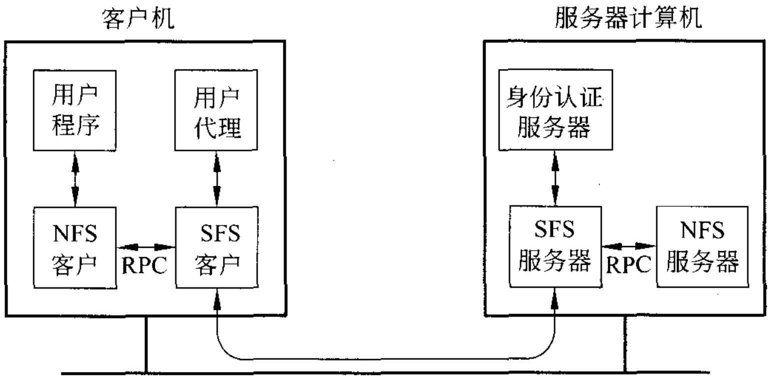


图 11.31 SFS 的组织结构

SFS 客户负责建立与 SFS 服务器通信的安全信道。它还负责与本地可用的 **SFS 用户代理**(SFS user agent)之间的通信,SFS 用户代理是一个自动处理用户身份认证的程序。SFS 并未规定如何进行用户身份认证。根据其设计目标,SFS 将这些身份认证问题分离出来,并且为不同的用户身份认证协议使用不同的代理。

在服务器端也有三个组件。使用 NFS 服务器也是出于可移植性的原因。NFS 服务器与 SFS 服务器通信,SFS 服务器是作为 NFS 服务器的 NFS 客户运行的。SFS 服务器构成 SFS 的核心进程。该进程负责处理来自 SFS 客户的文件请求。与 SFS 代理类似,SFS 服务器与一个单独的身份认证服务器通信来处理用户的身份认证。

与其他分布式文件系统相比,SFS 的独特之处在于它的名称空间的组织结构。SFS 提供一个全局名称空间,该名称空间以目录/sfs 为根。SFS 客户允许其用户在这个名称空间内创建符号链接。更重要的是,SFS 使用**自验证路径名**(self-certifying pathname)来命名它的文件。这种路径名实质上携带有用于验证由其命名的文件所在 SFS 服务器的身份的所有信息。自证明路径名由三部分组成,如图 11.32 所示。

路径名的第一部分由位置 LOC 组成,LOC 是标识 SFS 服务器的 DNS 域名或其对应的 IP 地址。SFS 假定每台服务器 S 都有一个公钥 K_S^+。自证明路径名的第二部分是主机标识符 HID,它是通过服务器的位置及其公钥上的加密散列 H 计算出来的:

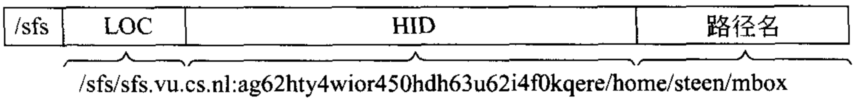


图 11.32 SFS 中的自证明路径名

$$HID = H(LOC, K_S^+)$$

HID 由基数为 32 的 32 位数表示。第三部分是实际存储文件的 SFS 服务器上的本地路径名。

每当客户访问 SFS 服务器时，它可以简单地通过向该服务器请求它的公钥来验证其身份。然后，客户可以使用众所周知的散列函数 H 计算 HID，并用该值与在路径名中获取的值进行比较。如果两个值匹配，那么客户就知道它正在与其名称与在位置部分中发现的名称相同的服务器通信。

这种方法如何把密钥管理和文件系统安全性分开呢？SFS 所解决的问题是：服务器的公钥的获取可以完全与文件系统安全性问题分开。一种获取服务器密钥的方法是让客户联系服务器并向其请求密钥，如上所述。不过，客户在本地存储一些密钥也是可以的，例如，由系统管理员存储。这样，客户就不需要联系服务器了。作为替代，在客户解析路径名时，在本地查找服务器的密钥，然后，它可以使用路径名中的位置部分来验证主机 ID。

为了简化问题，可以使用符号链接实现命名透明性。例如，假设客户想要访问下面这个文件：

```
/sfs/sfs.vu.cs.nl:ag62hty4wior450hdh63u62i4f0kqere/home/steen/mbox
```

为了隐藏主机 ID，用户可以创建一个符号链接：

```
/sfs/vucs→/sfs/sfs.vu.cs.nl:ag62hty4wior450hdh63u62i4f0kqere
```

然后，只使用路径名/sfs/vucs/home/steen/mbox。对该名称的解析将自动将该名称扩展为完整 SFS 路径名，然后，使用在本地找到的公钥，验证名称为 sfs.vu.cs.nl 的 SFS 服务器的身份。

与此类似，SFS 可以被证书颁发机构支持。通常，这种颁发机构会维护指向其代理的 SFS 服务器的链接。举例来说，考虑一个 SFS 证书颁发机构 CA，它运行如下 SFS 服务器：

```
/sfs/sfs.certsfs.com:kty83pad72qmbna9uefdppioq7053jux
```

假设客户已经安装了一个符号链接：

```
/certsfs→/sfs/sfs.certsfs.com:kty83pad72qmbna9uefdppioq7053jux
```

证书颁发机构可以使用另一个指向 SFS 服务器 sfs.vu.cs.nl 的符号链接：

```
/vucs→/sfs/sfs.vu.cs.nl:ag62hty4wior450hdh63u62i4f0kqere
```

这里，客户可以简单地引用/certsfs/vucs/home/steen/mbox，因为它知道它正访问的文件服务器的公钥已经被证书颁发机构 CA 认证。

回到分散式身份认证的问题上来，现在应该清楚，我们已经具有所有的机制，不需要在 Alice 的身份认证服务器上验证 Bob。如果给 Bob 的服务器提供了一个名称，Alice 就可以

简单地联系它。这个名称已经包含了一个公钥，使得 Alice 的服务器可以验证 Bob 服务器的身份。然后，Alice 的服务器可以按 Alice 所指示的那样接受 Bob 的特权。就像所说过的那样，可以在(Kaminsky 等 2003)中找到这种模式的详细信息。

11.8.3　安全的对等文件共享系统

迄今为止，我们讨论过的分布式文件系统都可以相对容易地保障其安全。传统的系统可以使用直观的身份认证以及利用安全通信进行扩展的访问控制机制，或者我们可以把传统的身份认证用于一种完全分散式的模式。不过，在涉及依靠协作的完全分散式系统(比如对等文件共享系统)时，事情将变得复杂。

11.8.3.1　基于 DHT 的系统中的安全查找

有多个问题需要处理(Castro 等 2002a；Wallach 2002)。让我们考虑基于 DHT 的系统。在这种情况下，我们需要依靠安全查找操作，它实质上可以归结为需要安全路由。这意味着：当无故障结点查找密钥 k 时，实际上将把它的请求转发给负责与 k 关联的数据的结点，或者转发给存储该数据的副本的结点。安全路由需要处理三个问题：

(1) 以安全的方式分配结点的标识符。

(2) 安全地维护路由表。

(3) 在结点之间安全地转发查找请求。

当没有给结点安全地分配其标识符时，我们可能面临如下问题：恶意结点可能给它自身分配一个 ID，使得所有查找特定密钥的请求都将被定向到它自身，或者沿着它属于其中一部分的路由转发。当多个结点协同工作时，这种情况将变得更严重，这实际上将允许一组结点构成许多查找请求的巨大的“汇点”。同样，如果没有安全的标识符分配，单个结点还能够给它自身分配许多标识符，这也称为**女巫攻击**(Sybil attack)，它可以造成相同的结果(Douceur 2002)。

与女巫攻击相比，更一般的攻击是：一个恶意结点控制如此多的无故障结点的邻居，以至于正确的结点几乎不可能正确地工作。这种现象也称为**日蚀攻击**(eclipse attack)，(Singh 等 2006)作过分析。你自己很难防御这类攻击。一种合理的解决方案是：限制每个结点的进入边(incoming edge)的数量。这样，攻击者可能只具有指向它的数量有限的正确结点。此外，为了阻止攻击者接管通往正确结点的所有进入链路，还应该限制外出链路的数量(Singh 等 2004)。所有这些情况下的问题都可以归结为：需要一个集中式颁发机构来分发结点标识符。显然，这样的颁发机构是与对等系统的分散式性质背道而驰的。

当可以用替代结点填充路由表时(在优化网络邻近性(network proximity)时经常这样做)，攻击者可以轻松地说服一个结点指向恶意结点。注意：当对填充路由表项提出了严格的限制条件时，就不会发生这种问题，在 Chord 中就是这样。因此，解决方案是把选择替代结点与更严格地限制填充路由表结合起来(Castro 等 2002a)。

最后，为了防御消息转发攻击，一个结点可以简单地沿着多条路由转发消息。执行该任务的一种方式是从不同的源结点启动查找。

11.8.3.2 安全协作存储

不过，要求结点之间进行协作这点琐事还引入了更多的问题。例如，协作可能规定结点应该提供它们从其他结点那儿使用的大约相同的存储空间。执行这种策略可能相当困难。一种解决方案是应用存储空间的安全交易，就像 Samsara 那样(Cox 和 Noble 2003)。

其思想相当简单：当服务器 P 想把它的文件之一 f 存储到另一个服务器 Q 上时，它将使其可用存储空间的大小等于 f 的大小，并且只为 Q 预留这个这间。换句话说，现在 Q 在 P 上具有一个未决**索赔**(claim)，如图 11.33 所示。

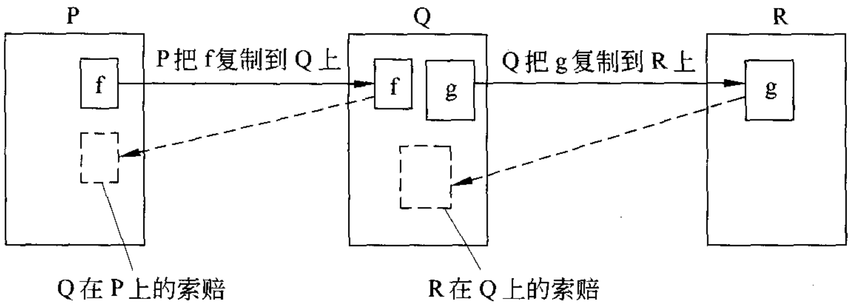


图 11.33 Samsara 对等系统中的存储空间索赔的原理

为了使这种模式正确地工作，每个参与者都要预留一定的存储空间，并把它分成大小相等的块。每个块中都包含不可压缩的数据。在 Samsara 中，块 c_i 包含 160 位的散列值 h_i，它是通过连接密码短语 W 与数字 i 计算得到的。现在假定在 256 字节的单元中分发索赔的存储空间。在这种情况下，将用前 12 个块以及下一个块的前 16 个字节计算第一个索赔的存储空间。将这些块连接起来，并使用私钥 K 对它们进行加密。一般来讲，索赔 C_j 计算如下：

$$C_j = K(h_k, h_{k+1}, \cdots, h_{k+11}, h_{k+12}[0], \cdots, h_{k+12}[15])$$

其中 $k=j\times 13$。无论何时 P 想利用 Q 上的存储空间，Q 都会返回 P 现在被迫存储的索赔集合。当然，Q 从不需要存储它自己的索赔，而可以根据需要计算它。

现在偶尔会遇到一点难以处理的事情，Q 可能想检查 P 是否仍在存储它的索赔。如果 P 不能证明它在这样做，Q 可能简单地丢弃 P 的数据。让 P 证明它仍然具有索赔的一种生硬的方式是把副本返回给 Q。显然，这将浪费许多带宽。假定 Q 把索赔 $C_{j_1}, \cdots, C_{j_k}$ 分发给 P。在这种情况下，Q 将把一个 160 位的串 d 传递给 P，并且请求它计算连接有 C_{j_1} 的 d 的 160 位散列值 d_1。然后把这个散列值与 C_{j_2} 相连接，产生散列值 d_2，依此类推。最终，P 只需返回 d_n 来证明它仍然保存有所有的索赔。

当然，Q 可能还想把它的文件复制到另一个结点(比如说 R)上。为了执行该操作，它将不得不保存 R 的索赔。不过，如果 Q 用光了存储空间，但它索赔了 P 上的存储空间，它可能决定把这些索赔传递给 R。这种原理的工作方式如下。

假定 P 保存了 Q 的索赔 C_Q，并且假定 Q 保存了 R 的索赔 C_R。由于没有限制 Q 可以在 P 上存储什么，Q 也可能决定在 P 上存储 C_R。然后，无论何时 R 想检查 Q 是否仍在保存它的索赔，R 都将把一个值 d 传递给 Q，并且请求它计算 d 与 C_R 连接的散列。为了执行该操作，Q 简单地把 d 传递给 P，请求 P 计算散列，并把结果返回给 R。如果它证明 P 不再保

存索赔，Q 将受到 R 的惩罚，Q 反过来又可以通过删除存储的数据来惩罚 P。

本章小结

分布式文件系统形成构建分布式系统的重要范型。它们通常按照客户-服务器模型组织，使用客户端缓存和对服务器复制的支持来满足可伸缩性的需求。另外，为实现高度可用性，需要使用缓存和复制。最近一段时间以来，又出现了对称式体系结构，比如对等文件共享系统中的那些体系结构。在这些情况下，就引出了一个重要的问题：是分布整个文件还是分布数据块。

通常不是直接把分布式文件系统构建在传输层之上，而是假定存在一个 RPC 层，使得所有的操作都可以简单地表示为文件服务器的 RPC，而不必使用原始的消息传递操作。目前已经开发了 RPC 的一些变体，比如 Coda 中提供的 MultiRPC，它允许并行调用许多服务器。

使分布式文件系统有别于非分布式文件系统的是共享文件的语义。理想情况下，文件系统可以使客户总是读取最近写入文件的数据。在分布式系统中实现这些 UNIX 文件共享语义是非常困难的。NFS 支持一种较弱形式的语义，称为会话语义，文件的最后版本取决于先前为写操作打开该文件并最后关闭该文件的客户。在 Coda 中，读数据的客户只有重新打开文件时才能看到最新的更新数据，从这种意义上说，文件共享遵循事务处理语义。Coda 的事务处理语义不涉及常规事务处理的所有 ACID 属性。在服务器控制所有操作的情况下，可以提供实际的 UNIX 语义，但是这样会造成可伸缩性问题。在所有情况下，都需要允许对文件进行并发更新，它使相对复杂的锁定和预约模式开始派上用场。

为了实现可接受的性能，分布式文件系统通常允许客户缓存整个文件。例如，NFS 支持这种完整文件缓存的方法，尽管它也可能只存储文件的大部分数据。一旦文件已经打开，并(部分地)传送给客户，则所有操作都在客户本地执行。当文件再次被关闭时，客户向服务器返回所有的更新。

复制在对等系统中也扮演着重要角色，尽管由于文件一般是只读的而使得事情得到了极大的简化。在这些系统中，更重要的是尝试达到可接受的负载平衡，因为以前未试验过的复制模式可能轻松导致保存许多文件的热点，从而变成潜在的瓶颈。

在使用传统方法时，通常还要考虑容错性。不过，也有可能构建可以处理 Byzantine 故障的文件系统，甚至当整个系统都运行在因特网上时。在这种情况下，通过假定合理的超时时间并启动新的服务器组(可能基于故障检测)，可以构建实用的解决方案。特别是对于分布式文件系统来说，当实现高度可用性是唯一目标时，人们应该考虑应用擦除编码技术来减小总复制因子。

安全性是任何分布式系统(包括文件系统)的最重要的方面。NFS 本身几乎不提供任何安全性机制，而代之以实现标准化接口，这使得它可以使用不同的现有安全性系统，比如 Kerberos。与之不同的是，SFS 允许文件名包含文件服务器的公钥的信息。这种方法简化了大规模系统中的密钥管理。实际上，SFS 通过将密钥包含于文件的名称之中来分发密钥。SFS 可以用于实现分散式身份认证模式。在对等文件共享系统中实现安全性很困难，这部分是由于在假定的协作性中结点总是倾向于是自私的。此外，保证查找安全被证明是一个困难的问题，它实际上需要中心颁发机构来分发结点标识符。

习题

1. 实现 NFS 版本 3 的文件服务器必须是无状态的吗?
2. 解释是否可以把 NFS 视作一种分布式文件系统。
3. 尽管 GFS 的可伸缩性很好,但是可以证明主服务器仍然是一个潜在的瓶颈。可以用什么合理的技术替换它?
4. 使用 RPC2 的副作用处理连续的数据流很方便。请给出另一个示例,在该例子中使用一种仅次于 RPC 的针对应用程序的协议是有意义的。
5. NFS 没有提供全局共享的名称空间。是否有模拟这种名称空间的方法?
6. 请简单扩展 NFS 的 lookup 操作,结合输出从另一个服务器挂载的目录的服务器,使它允许进行迭代式名称查找。
7. 在基于 UNIX 的操作系统中,使用文件句柄打开文件的操作只能在内核中实现。对于 UNIX 系统的用户级 NFS 服务器,请给出 NFS 文件句柄的可能实现。
8. 如本章所述,自动挂载器安装符号链接,使用这样的自动挂载器导致更难以隐藏挂载是透明的这一事实。为什么?
9. 假设 NFS 中一个文件的当前拒绝状态是 WRITE。另一个客户是否可以先成功地打开该文件,然后请求一个写操作锁?
10. 考虑第 7 章所讨论的缓存相关性,说明 NFS 实现的是哪种缓存相关性协议?
11. NFS 是否实现入口一致性?
12. 我们陈述了 NFS 实现文件处理的远程访问模型。我们可以认为它也支持上载/下载模型。请解释其原因。
13. NFS 使用直写式缓存相关性策略缓存属性。它是否有必要立即转发所有的属性更改?
14. 出现故障时,RPC2 提供什么调用语义?
15. 如果一个文件在多个读文件的人和唯一一个写文件的人之间共享,解释 Coda 如何解决该文件上的读-写冲突。
16. 使用自证明路径名,是否总能保证客户与非恶意的服务器通信?
17. (实验)构建基于 UNIX 的分布式文件系统的最容易的方式之一是:通过 NFS 把许多机器连接起来。对于这道作业题,你将通过 NFS 连接不同计算机上的两个文件系统。具体地讲,在一台机器上安装 NFS 服务器,使得在第一台机器启动时将会自动挂载其文件系统的多个部分。
18. (**实验**)为了把基于 UNIX 的机器与 Windows 客户相集成,可以利用 Samba 服务器。扩展上一道作业题,通过安装和配置 Samba 服务器,使基于 UNIX 的系统可供 Windows 客户使用。与此同时,应该保持文件系统可以通过 NFS 访问。

第12章

基于Web的分布式系统

可以把**WWW**(World Wide Web,万维网)视为一个巨大的分布式系统,其中包含数以百万计的客户和服务器用于访问链接文档。服务器维护文档的集合,而客户为用户提供可以表示和访问这些文档的易于使用的界面。

Web起源于位于日内瓦的欧洲**粒子物理研究所**(European Particle Physics Laboratory,CERN)的一个项目,其目的是使其地理位置上分散的众多研究者能够使用一种简单的超文本系统来访问共享文档。文档可以是任何能显示在用户的计算机终端上的内容,例如个人便笺、报告、图形、蓝图和图画等。通过相互链接文档,无须集中更改就很容易把来自不同项目的文档集成到一个新文档中。只需构建一个提供到其他相关文档的链接的文档即可(Berners-Lee等 1994)。

逐渐地,Web中的站点不再局限于高能物理领域,而是扩展了到世界范围。但是Web真正得到普及是在出现像Mosaic(Vetter等 1994)这样的图形用户界面之后。Mosaic提供了只需单击鼠标即可表示和访问文档的易用界面。通过这种界面,可以从服务器上获取文档,把它传递给客户,然后显示在屏幕上。对于用户而言,本地存储的文档和存储在世界上其他地方的文档没有什么概念上的区别。从这种意义上讲,分布是透明的。

从1994年起,Web的发展主要由**万维网联盟**(World Wide Web Consortium)发起并控制,该联盟由CERN和M.I.T.(美国麻省理工学院)合办,负责制订协议标准、改善互操作性和进一步增强Web的能力。此外,我们还可以看到这个联盟之外的许多新发展,它们并不总是像人们期望的那样相互兼容。目前,Web已经超越了简单的基于文档的系统这个范畴。特别是在引入Web服务之后,我们看到出现了巨大的分布式系统,在这种系统中使用和创建的是**服务**(service),而不仅仅是文档,并把它们提供给任何可以发现它们的用户和机器。

在本章中,我们将详细介绍这种快速发展、普及的系统。考虑到Web本身刚出现不久,并且在如此短的时间内发生了如此多的变化,因此我们只能粗略描述它的当前状态。不过,如我们将看到的,Web技术底下的许多概念都基于在本书第一部分中讨论过的原理。此外,我们还会看到,对于许多概念来说,还有很大的改进空间。

12.1 体系结构

基于Web的分布式系统的体系结构与其他分布式系统并没有根本的区别。不过,支持分布式文档的初始思想于20世纪90年代出现以来是如何演进的呢?查明这一点是有趣

的。这些文档从纯粹静态的被动式文档转变成动态生成的文档，其中包含各种活动元素。此外，近年来，许多组织开始支持服务，而不仅仅是文档。接下来，我们将讨论这些变化对体系结构带来的影响。

12.1.1 传统的基于 Web 的系统

与我们迄今为止讨论过的许多分布式系统不同，基于 Web 的分布式系统相对较新。从这种意义上讲，讨论传统的基于 Web 的系统有点困难，尽管在最初可用的系统与今天使用的系统之间泾渭分明。

许多基于 Web 的系统仍然被组织为相对简单的客户-服务器体系结构。一个可以访问存储文档的本地文件系统的进程构成了 Web 站点的核心。指定文档的最简单的方式是使用一个称为**统一资源定位符**（Uniform Resource Locator，URL）的引用。它指定了文档所在的位置，通常采用的方法是嵌入与文档关联服务器的 DNS 名称以及一个文件名，服务器可以在其本地文件系统中根据这个文件名查找文档。而且，URL 还指定了在网络上传送文档所使用的应用级协议。可用的协议有多种，我们下面会对它们进行解释。

客户与 Web 服务器通过一个称为**浏览器**（browser）的特殊应用程序进行交互。浏览器负责正确地显示文档。此外，浏览器还负责接受用户的输入，通常是让用户选择对另一个文档的引用，然后去获取并显示所选的文档。浏览器与 Web 服务器之间的通信方式已经标准化：它们都遵守**超文本传输协议**（HyperText Transfer Protocol，HTTP），我们将在下面讨论它。这导致了如图 12.1 所示的常见组织结构。

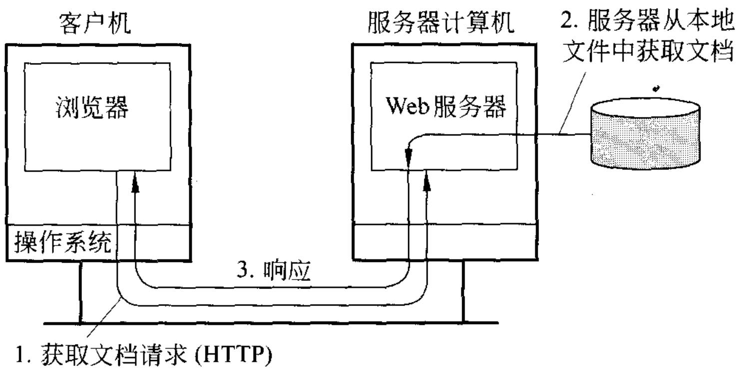


图 12.1 传统 Web 站点的常见组织结构

自从 Web 出现以来，它已经取得了相当可观的进步。到现在为止，已开发出大量的方法和工具用于产生可以由 Web 客户和 Web 服务器处理的信息。下面，我们将深入讲述 Web 作为一个分布式系统的工作原理。不过，我们将不讨论用于构造 Web 文档的大部分方法和工具，因为它们通常和 Web 的分布式特性没有直接的关系。（Sebesta 2006）有关于如何构建基于 Web 的应用程序的详尽介绍。

12.1.1.1 Web 文档

Web 的基础是几乎所有的信息都以文档的形式表示。从广义上讲：文档不仅可以包含纯文本，而且可以包括各种动态特性，比如音频、视频、动画等。在许多情况下，需要特殊的

辅助应用程序使文档“具有生命”。这样的解释器通常与用户的浏览器集成在一起。表达文档的方式是多种多样的。有些文档像ASCII文本文件一样简单，而有些文档由脚本集合来表达，当文档被下载到浏览器时这些脚本会自动执行。

大多数文档可以粗略地分为两个部分：主要部分和次要部分，前者至少充当后者的模板。主要部分包含许多不同的位和信息块，它们共同构成了在浏览器中显示的文档。主要部分一般是用**置标语言**(markup language)编写的，它非常类似于在字处理系统中使用的语言类型。Web中最广泛使用的置标语言是**超文本置标语言**(HyperText Markup Language,HTML)。顾名思义，HTML允许嵌入对其他文档的链接。当在浏览器中激活这种链接时，将从其关联的服务器上获取所引用的文档。

另一种日益重要的置标语言是**可扩展置标语言**(Extensible Markup Language,XML)，顾名思义，它在定义文档外观方面提供了更大的灵活性。HTML与XML之间的主要区别是：后者包括标记文档的元素的定义。换句话说，它是一种元置标语言。在精确指定文档看起来像什么时，这种方法提供了许多灵活性：无需遵守一种固定的置标语言(如HTML)规定的单独一种模型。

HTML和XML也可以包括各种称为**嵌入式文档**(embedded document)的标签，它们是对一些文件的引用，为了使文档完整而应该包括这些文件。可以证明：嵌入式文档把Web文档转变成活动文档。尤其是考虑到嵌入式文档可以是作为显示信息的一部分而动态执行的完整程序，就不难想象它可以做什么事情。

嵌入式文档具有各种类型和风格。这立即引出了一个问题：如何配置浏览器以处理不同的文件格式以及解释嵌入式文档的不同方式。实质上讲，我们只需要两件事：指定嵌入式文档类型的方式和允许浏览器处理特定类型的数据的方式。

每个(嵌入式)文档都具有一个关联的**MIME类型**(MIME type)。MIME代表**多用途网际邮件交换**(Multipurpose Internet Mail Exchange)，顾名思义，最初开发它时，是为了提供关于作为电子邮件一部分发送的消息体内容的信息。MIME区分多种不同类型的消息内容。这些类型也用在WWW中，但是人们注意到难以对最近出现的新数据格式进行标准化。

MIME中的类型分为顶级类型和子类型。图12.2中显示了一些常见的顶级类型，包括用于文本、图像、音频和视频的类型。还有一种特殊的application类型，它表示这种类型的文档中包含的数据与特定的应用程序相关。实际上，只有这个应用程序才能把文档转换为可以被人们理解的内容。

multipart类型用于合成文档。合成文档由多个部分组成，每个部分还可以有各自关联的顶级类型。

每种顶级类型都可以有多种子类型，其中一些如图12.2所示。文档类型由顶级类型和子类型的组合来表示，例如application/PDF。这种文档类型说明需要一个独立的应用程序来处理文档，并且该文档以PDF格式表示。许多子类型还处在试验阶段，这意味着在使用特殊格式时需要用户端具有它自己的应用程序。实际上，由Web服务器提供这种应用程序，它可以是在浏览器旁边运行的单独的应用程序，或者是可以被安装为浏览器一部分的所谓的**插件**(plug-in)。

这种(变化的)文档类型迫使浏览器必须是可扩展的。最终，开展了一些标准化工作，允

类型	子类型	说明
text	Plain	未格式化的文本
	HTML	包含 HTML 标记命令的文本
	XML	包含 XML 标记命令的文本
image	GIF	GIF 格式的静态图像
	JPEG	JPEG 格式的静态图像
audio	Basic	音频，8位 PCM 编码，采样率为 8000 Hz
	Tone	一种特殊的可听见的音调
video	MPEG	MPEG 格式的影片
	Pointer	用于展示的定位设备的表示法
application	Octet-stream	未解释的字节序列
	Postscript	可打印的 Postscript格式的文档
	PDF	可打印的 PDF 格式的文档
multipart	Mixed	以指定顺序排列的独立部分
	Parallel	必须同时查看的部分

图 12.2　6 种顶级 MIME 类型和一些常见的子类型

许在浏览器中轻松地集成遵守某些接口的插件。当某些类型变得足够普及时，通常将与浏览器或其更新一些提供它们。在下面讨论客户端软件时，将回过头来探讨这个问题。

12.1.1.2　多层体系结构

HTML（或其他任何置标语言，如 XML）与脚本的结合提供了表达文档的一种强大手段。不过，我们还没有讨论过文档实际上在哪里进行处理和进行什么样的处理。WWW 开始时是一个如前面图 12.1 所示的相对简单的客户-服务器系统。但现在，这种简单的体系结构已扩展到包含众多组件以支持我们刚才描述过的那些高级文档类型。

对基本体系结构的主要增强之一是通过**公共网关接口**（Common Gateway Interface，CGI）支持简单的用户交互。CGI 定义了 Web 服务器把用户数据作为输入来执行程序的标准方式。通常，用户数据来自 HTML 表单，它指定了要在服务器端执行的程序，以及由用户填写的参数值。一旦完成了表单，程序的名称和从用户处收集的参数值会被发送到服务器，如图 12.3 所示。

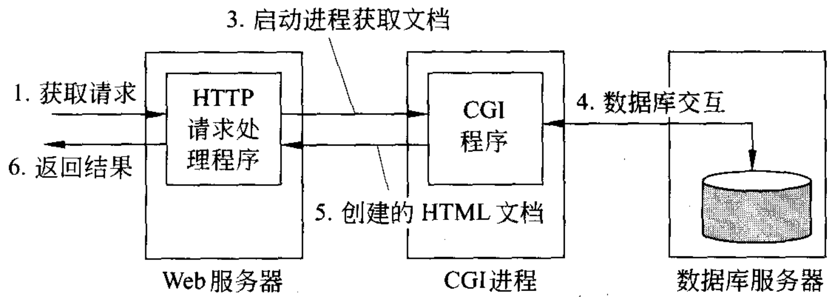


图 12.3　使用服务器端 CGI 程序的原理

当服务器收到请求时，它启动请求中指定的程序，并把参数值传递给该程序。此时，程序只是完成它自己的工作，并且通常以文档的形式返回结果，该文档会被发送回用户的浏览

器以进行显示。

只要开发人员需要，CGI 程序可以很复杂。例如像图 12.3 中那样，许多程序对位于 Web 服务器本地的数据库进行操作。在处理完数据后，程序生成 HTML 文档并把它返回给服务器。随后服务器把文档传递给客户。有趣的是，对于服务器而言，这就像是它要求 CGI 程序获取文档。也就是说，服务器所做的只是把获取文档的任务委托给一个外部程序。

服务器的主要任务原先只是简单地获取文档来处理客户请求。有了 CGI 程序，服务器只需把获取文档的任务委托给这类程序即可，它不需要知道文档是动态生成的，还是从本地文件系统中实际读出的。注意：我们刚才描述了服务器端软件的一个两层组织结构。

然而，今天的服务器除了获取文档之外，还要完成许多其他任务。最重要的增强功能之一是：服务器在把获取的文档传递给客户之前还可以对它进行一些处理。特别是，一个文档也许会包含一段**服务器端脚本**(server-side script)。在从本地获取该文档之后，服务器将会执行这段脚本，然后将把脚本执行的结果与文档的其余部分一起发送给客户。但脚本本身不会被发送。也就是说，使用服务器端脚本修改文档实质上是通过用脚本的执行结果替换脚本来完成的。

Web 文档的服务器端处理越来越需要更大的灵活性，无疑，现在许多 Web 站点都被组织为三层体系结构，它们由 Web 服务器、应用程序服务器和数据库组成。Web 服务器是我们以前所具有的传统 Web 服务器；应用程序服务器运行各类可能会或者可能不会访问第三层的程序，第三层由数据库组织。例如，服务器可能接受顾客的查询，搜索匹配产品的数据库，然后构造一个可单击的 Web 页面，列出找到的产品。在许多情况下，服务器负责运行 Java 程序(称为 **servlet**)，它们用于维护像购物车之类的事物、实施建议、保留最喜爱的商品列表等。

不过，这种三层组织结构引入了一个问题：它会导致性能下降。尽管从体系结构的角度讲分为三个层是有意义的，但是实践表明应用程序服务器和数据库是潜在的瓶颈。特别是改进数据库性能可能被证明是一个难以处理的问题。在下面讨论将缓存和复制作为性能问题的解决方案时，我们将回过头来探讨这个问题。

12.1.2 Web 服务

迄今为止，我们隐式地假定基于 Web 的系统的客户端软件由浏览器组成，浏览器充当用户的接口。这种假定并不是放之四海而皆准的。有一类快速增长的基于 Web 的系统，它们给远程应用程序提供了常规的服务，而无需最终用户的即时交互。这种组织结构引出了 **Web 服务**(Web service)的概念(Alonso 等 2004)。

12.1.2.1 Web 服务的基本原理

简单地讲，Web 服务只是可以通过因特网使用的传统服务(例如，命名服务、天气报告服务、电子供应商等)。使 Web 服务与众不同的是它遵守一个标准集合，它允许也遵守这些标准的客户应用程序通过因特网发现和访问 Web 服务。这样，这些标准无疑就构成了 Web 服务体系结构的核心(Booth 等 2004)。

提供和使用 Web 服务的原理相当简单，如图 12.4 所示。其基本思想是：某个客户应

用程序可以调用服务器应用程序提供的服务。目前已经开发了一些关于如何描述这些服务的标准，以便客户应用程序可以查找它们。此外，我们需要确保服务调用将遵循服务器应用程序设置的规则进行处理。注意：这种原理不同于实现远程过程调用所需的原理。

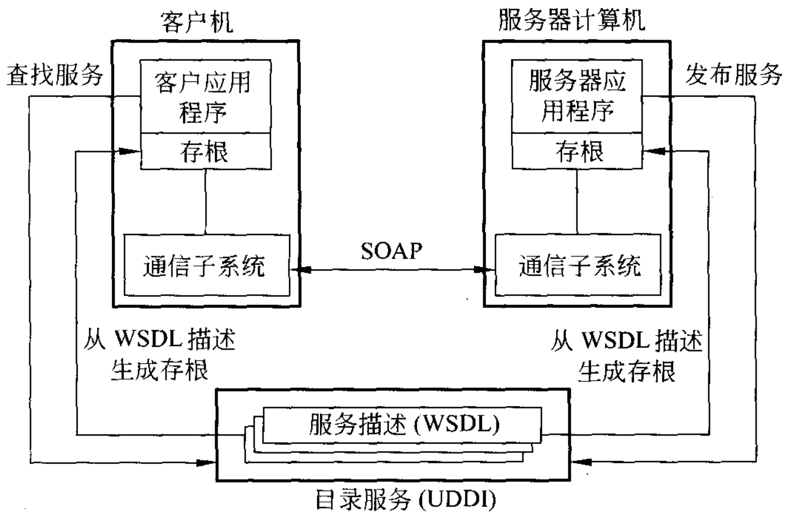


图 12.4 Web 服务的原理

Web 服务体系结构中的一个重要组件是由存储服务描述的目录服务构成的。这种服务遵守**通用描述、发现和集成**(Universal Description, Discovery and Integration, UDDI)标准。顾名思义，UDDI 规定了包含服务描述的数据库的布局，这些服务描述将允许 Web 服务客户浏览相关的服务。

通过 **Web 服务定义语言**(Web Services Definition Language, WSDL)来描述服务，该语言是一种形式语言，非常类似于用于支持基于 RPC 的通信的接口定义语言。WSDL 描述包含服务提供的接口的精确定义，即过程规范、数据类型和服务的（逻辑）位置等。WSDL 描述的一个重要问题是：它可以被自动转换成客户端和服务器端存根，这也类似于基于 RPC 的普通系统中的存根的生成方式。

最后，Web 服务的核心元素是通信如何发生的规范。此时，将使用**简单对象访问协议**(Simple Object Access Protocol, SOAP)，它实质上是一种框架，在其中可以标准化两个进程之间的大部分通信。我们将在下面详细讨论 SOAP，同时我们还会清楚调用框架很简单，而根本无需加以证明。

12.1.2.2 Web 服务的合成与协调

迄今为止描述的体系结构相对比较直观：通过应用程序实现服务，并根据特定的标准来调用它。当然，应用程序本身可能很复杂，事实上，其组件可能是在整个局域网中完全分布的。在这种情况下，最有可能通过内部代理或守护进程实现 Web 服务，该内部代理或守护进程与构成分布式应用程序的多种组件交互。在这种情况下，可以像我们所讨论的那样很容易地应用我们迄今为止讨论过的所有原理。

在迄今为止的模型中，Web 服务是以单个调用的形式提供的。实际上，在可以认为服务完成之前，需要采用复杂得多的调用结构。以一家电子书店为例。订购一本图书需要选择一本图书、付款以及确保其交货。从服务的角度讲，实际的服务应该被建模为包含多个步

骤的事务，需要以特定的顺序执行这些步骤。换句话说，我们正在与**复杂服务**(complex service)打交道，它是从许多基本服务构建而成的。

当考虑通过组合来自不同提供者的Web服务来提供Web服务时，情况就变得复杂了。典型的示例是设计一家基于Web的商店。大多数商店大体上由三部分组成：通过第一部分来选择客户所需的商品，第二部分处理这些商品的付款，第三部分负责送货和后续的商品跟踪。为了建立这样一家商店，提供者可能希望利用可以处理付款的电子银行服务，还可能希望利用处理送货的特殊交货服务。这样，提供者就可以集中关注其核心业务，即提供商品。

在这些场景中，重要的是让顾客看到一种条理分明的服务：即他可以在这家商店选择商品、付款以及依靠正确的交货。不过，我们需要在内部处理如下情形：可能有三个不同的组织需要以一种协调一致的方式运作。提供对这种**复合服务**(composite service)的正确支持构成了Web服务的必要元素。至少有两类问题需要解决。第一，如何在可能来自不同组织的Web服务之间进行协调？第二，怎样才能轻松地组成服务？

可以通过**协调协议**(coordination protocol)来处理Web服务之间的协调任务。这种协议规定了为了让(复合)服务取得成功而需要执行的多个步骤。当然，问题是要强制参与这种协议的各方在正确的时刻采取正确的步骤。可以用多种方式实现这一点；最简单的方式是使用单个协调器来控制参与各方之间交换的消息。

不过，尽管存在多种解决方案，但是从Web服务的角度讲，重要的是标准化协调协议中的共性。举例来说，当一方想参与特定的协议时，它知道它应该与哪些其他的进程通信是重要的。此外，让一个进程同时参与到多个协调协议中也可能很好。因此，确定协议的实例也很重要。最后，一个协议应该知道它将履行哪种角色。

在称为**Web服务协调**(Web Services Coordination)的技术中标准化了这些问题(Frend等2005)。从体系结构的角度讲，它定义了用于处理协调协议的单独的服务。协调协议是这种服务的一部分。进程可以把它们自身注册为参与协调，使得其对应的进程能够知道它们。

为了使事情具体一些，考虑用于我们在第8章中讨论过的两阶段协议(2PC)的变体的协调服务。其总体思想是：这样一种服务将实现用于多种协议实例的协调器。一种显而易见的实现是：单个进程为多个协议实例扮演协调器的角色。一种替代方案是由单独的线程实现每个协调器。

进程可以请求激活特定的协议。此时，实质上将给它返回一个标识符，它可以把该标识符传递给其他进程，以便注册为最近创建的协议实例中的参与者。当然，所有参与的进程都将被要求实现协调服务支持的协议的特定接口。一旦所有参与者都已注册，协调器就可以根据需要把VOTE_REQUEST、COMMIT以及属于2PC协议一部分的其他消息发送给参与者。

不难看出：由于(例如)2PC协议中的共性，要交换的接口和消息的标准化使得更容易组成和协调Web服务。需要做的实际工作并不是很困难。从这个方面讲，在标准化中完全可以看出协调服务的附加价值。

显然，协调服务已经为从其他服务中组成Web服务提供了便利。只有一个潜在的问题：如何组成服务是公开的。在许多情况下，这不是人们期望的性质，因为它将允许任何竞

争者建立完全相同的复合服务。因此，我们需要的是为建立私有协调器提供便利。我们将不会在这里深入讨论任何细节，因为它们没有触及基于 Web 的系统中的服务合成的原理。此外，这种合成仍然在不断发生很大的变化(并且可能会在很长一段时间内继续如此)。有兴趣的读者可以参阅 Alonso 等人的著作(2004)。

12.2 进程

我们现在转向介绍基于 Web 的系统中使用的最重要的进程以及它们的内部组织结构。

12.2.1 客户

最重要的 Web 客户是一种称为 **Web 浏览器**(Web browser)的软件，它通过从服务器获取 Web 页面并把它们显示在用户的屏幕上来允许用户访问这些页面。浏览器一般提供带有超链接的界面，用户只需单击一个超链接就可以轻松地选中它。

Web 浏览器过去是简单的程序，但这是很多年前的事了。从逻辑上讲，它们包含多种组件，如图 12.5 所示(Grosskurth 和 Godfrey 2005)。

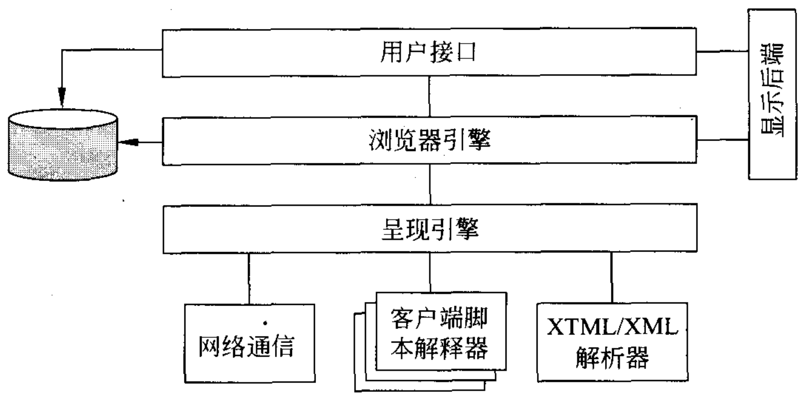


图 12.5 Web 浏览器的逻辑组件

Web 浏览器的一个重要方面是：(在理想情况下)它们应该是与平台无关的。通过利用标准图形库(如图 12.5 中所示的显示后端)以及标准网络库通常可以实现这个目标。

浏览器引擎和呈现引擎构成了浏览器的核心。如我们以前所解释的，呈现引擎包含用于正确显示文档的所有代码。这种呈现至少需要解析 HTML 或 XML，可能还需要脚本解释。在大多数情况下，只包含一个用于 JavaScript 的解释器，但是从理论上讲也可能包括其他解释器。浏览器引擎提供了一些机制，允许最终用户查看文档、选择其中的一部分、激活超链接等。

Web 浏览器的设计者需要面对的一个问题是浏览器应该很容易扩展，以使它从原理上讲可以支持服务器返回的任何类型的文档。大多数浏览器采用的方法是提供一种称为插件的功能。**插件**(plug-in)是可以动态加载到浏览器中以便处理某种特定类型文档的小型程序。文档类型一般作为 MIME 类型给出。插件应该在本地可用，有可能需要事先由用户专

门从远程服务器传输到本地。插件通常向浏览器提供一个标准的接口，并且也希望浏览器为它提供一个标准接口。从逻辑上讲，它们构成了图12.5中所示的呈现引擎的扩展。

另一个常用的客户端进程是**Web代理**(Web proxy)(Luotonen和Altis 1994)。起初，这种进程用于允许浏览器处理不同于HTTP的应用级协议，如图12.6所示。例如，要从一个FTP服务器传输文件，浏览器可以发送一个HTTP请求给本地FTP代理，后者将获取文件并把它嵌入在一个HTTP响应消息中返回给浏览器。



图12.6 当浏览器不支持FTP时就使用Web代理

现在，大多数Web浏览器都能够支持多种协议，或者可以动态地扩展到具有这种能力，因而不再需要使用代理。不过，出于其他原因仍在使用Web代理。例如，可以将代理配置成过滤请求和响应(使之接近于应用级防火墙)、进行日志记录、压缩数据，但是其中大多数都是缓存的。我们会在下面回过头来讨论缓存代理。一个广泛使用的Web代理是**Squid**，它是作为一个开源项目开发的。可以在(Wessels 2004)中找到关于Squid的详细信息。

12.2.2 Apache Web服务器

迄今为止最流行的Web服务器是Apache，估计在全部Web站点中使用Apache Web服务器的大约有70%。Apache是一种复杂的软件，并且具有众多增强功能，可以处理Web中现在提供的各类文档，重要的是使服务器是高度可配置和可扩展的，同时在很大程度上与特定的平台无关。

为了使服务器与平台无关，实质上是通过提供它自己的基本运行库环境，然后可以为不同的操作系统实现这个运行库环境。该环境(称为Apache可移植运行库(Apache Portable Runtime，APR))是一个库，它提供了与平台无关的接口，用于文件处理、联网、锁定、线程等。在扩展Apache时(我们稍后将讨论)，一般通过假定只调用APR而避免调用特定于平台的库，来保证可移植性。

如我们所说过的，Apache被定制成不仅提供了灵活性(从可以在相当程度上配置它这个角度讲)，而且还能相对容易地扩展其功能。例如，在本章后面我们将讨论Globule中的自适应性复制，Globule是由位于阿姆斯特丹自由大学(Vrije Universiteit Amsterdam)的创作组开发的一种自制的内容交付网络。Globule被实现为Apache的一种扩展，它基于APR，而且在很大程度上与为Apache开发的其他扩展无关。

从某种角度讲，可以把Apache看作一种完全普通的服务器，它被定制成对进入的请求产生一条响应。当然，还有各种隐藏的相关性和假定，由此证明Apache主要适合于处理对Web文档的请求。例如，如前所述，Web浏览器和服务器使用HTTP作为它们的通信协议。HTTP几乎总是在TCP之上实现的，因此，Apache的核心假定所有进入的请求都遵守基于TCP的面向连接的通信方式。例如，如果不修改Apache核心，将不能正确地处理基于UDP的请求。

不过，Apache核心对应该如何处理进入的请求做出了几条假定。其总体组织结构如图12.7所示。这种组织结构的基本原理是**挂钩**(hook)的概念，它只是一组特定函数的占位符。Apache核心假定请求在许多阶段中处理，其中每个阶段都由几个挂钩组成。因此，每个挂钩代表需要作为处理请求的一部分而执行的一组类似的动作。

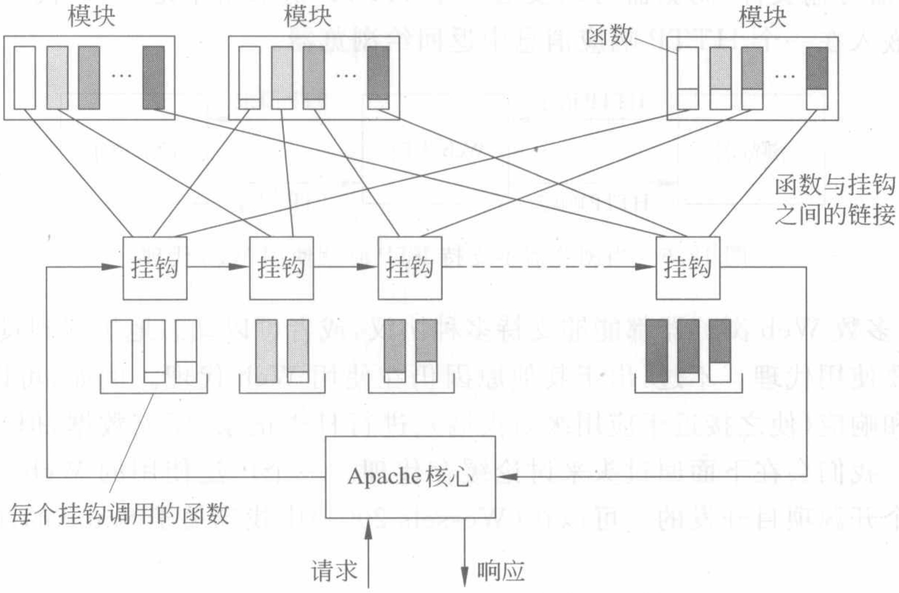


图12.7　Apache Web服务器的一般组织结构

例如，有一个挂钩用于把URL转换成一个本地文件名。在处理请求时，几乎肯定需要进行这种转换。此外，还有用于把信息写到日志中的挂钩、用于检查客户标识的挂钩、用于检查访问权限的挂钩，以及用于检查与请求相关的MIME类型的挂钩(例如，确保可以正确地处理请求)。如图12.7所示，挂钩是以预先确定的顺序处理的。我们在这里可以明确看出：Apache执行了关于请求处理的特定控制流。

与一个挂钩关联的函数全都是由单独的**模块**(module)提供的。尽管从原理上讲开发人员可以改变将由Apache处理的挂钩集，但是更常见的做法是编写包含一些函数的模块，在处理由未经修改的Apache提供的标准挂钩时，将需要调用这些函数。其基本原理相当直观。每个挂钩可以包含一组函数，其中每个函数都应该匹配一个特定的函数原型(即参数列表和返回类型)。模块开发人员将为特定的挂钩编写函数。在编译Apache时，开发人员指定应该把哪个函数添加到哪个挂钩中。在图12.7中，把后者显示为函数与挂钩之间的多种链接。

由于可能有数十种模块，每个挂钩一般将包含几个函数。通常，模块被看作是相互独立的，以便同一个挂钩中的函数可以以任意的顺序执行。不过，Apache也可以处理模块相关性，其方法是让开发人员指定处理不同模块中的函数的顺序。总的说来，结果是使Web服务器变得极其通用。在(Laurie和Laurie 2002)中可以找到关于配置Apache的详细信息，以及如何扩展它的良好指导。

12.2.3 Web服务器群集

与Web的客户-服务器特性有关的一个重要问题是Web服务器很容易过载。为许多设计所采用的一个实用解决方案是：简单地把一个服务器复制到一个服务器群集中，并使用单独的机制（如一个前端）把客户的请求重定向到这些副本服务器之一。其原理如图12.8所示，这是我们在第2章中讨论的水平分布的一个示例。

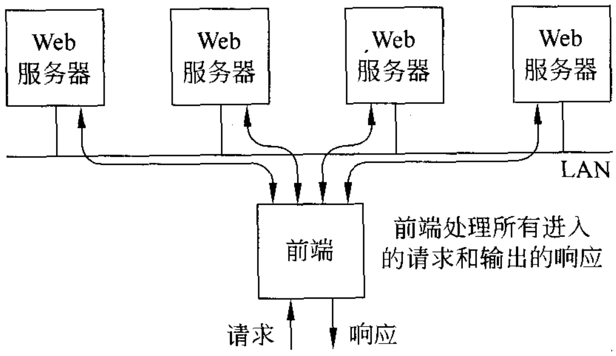


图12.8 结合使用服务器群集与前端来实现Web服务的原理

这种组织结构的一个关键方面是前端的设计，因为前端很容易成为一个严重的性能瓶颈，所有的通信量都要经过它。一般来讲，作为传输层交换机运行的前端与运行在应用层级别的前端有很大的区别。

无论何时客户发出一个HTTP请求，它都需要和服务器建立一个TCP连接。一个传输层交换机根据对服务器负载的某种度量，简单地把通过TCP连接发送的数据传递给服务器之一。把来自服务器的响应返回给交换机，交换机然后把它转发给请求的客户。作为一种优化方法，可以使交换机和服务器合作来实现**TCP移交**（TCP handoff），如我们在第3章中所讨论的那样。传输层交换机的主要缺点是：交换机不能考虑通过TCP连接发送的HTTP请求的内容，它只能基于服务器负载情况做出重定向决定。

一般来说，更好的方法是采用**内容感知的请求分布**（content-aware request distribution）。使用这种方法，前端首先检查进入的HTTP请求，然后决定应当把该请求转发给哪个服务器。内容感知的分布方法具有几个优点。例如，如果前端总是把对同一个文档的请求转发给同一个服务器，该服务器就能有效地缓存这个文档，从而缩短响应时间。此外，它还可能实际地在服务器当中分布文档集合，而不必为每个服务器复制每个文档。这种方法更高效地利用了可用的存储空间，并且允许使用专用服务器来处理特殊的文档，如音频和视频。

内容感知的分布方法的一个问题是前端需要做很多工作。在理想情况下，人们希望具有TCP移交的效率和内容感知的分布方法的功能。我们需要做的就是分布前端的工作，并结合使用传输层交换机，如（Aron等2000）中所提议的那样。在结合使用TCP移交的情况下，前端有两个任务。首先，当一个请求第一次到来时，前端必须决定由哪个服务器来处理与客户的剩余通信。其次，前端应该转发与被移交的TCP连接相关联的客户的TCP消息。

这两个任务的分布如图12.9所示。**调度器**（dispatcher）负责决定应该把一个TCP连

接移交给哪个服务器；**分发器**(distributor)监视一个被移交的连接中进入的 TCP 通信量。交换机用来把 TCP 消息转发给分发器。当一个客户首次联系 Web 服务时，将把它的 TCP 连接建立消息转发给一个分发器；分发器继而与调度器联系，由后者决定应把该连接移交给哪个服务器。此时，交换机会得到通知，把该连接的所有后续 TCP 消息发送给所选的服务器。

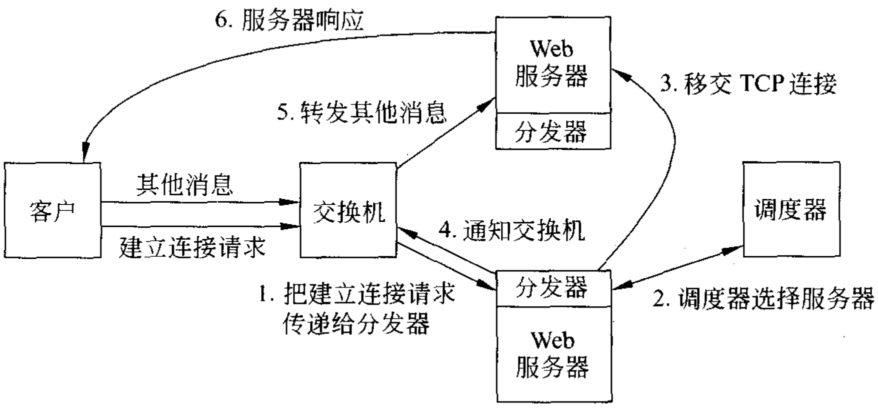


图 12.9 一个可伸缩的内容感知的 Web 服务器群集

有多种其他的替代方法可用于建立 Web 服务器群集并对它们进行进一步的细化。例如，无需使用任何类型的前端，也可能使用**循环 DNS**(round-robin DNS)，通过它把单个域名与多个 IP 地址关联起来。这样，在解析 Web 站点的主机名称时，客户浏览器将接收到多个地址的列表，其中每个地址都对应于 Web 服务器之一。通常，浏览器会选择列表上的第一个地址。不过，像 BIND 这样流行的 DNS 服务器将循环它返回的列表中的条目(Albitz 和 Liu 2001)。因此，我们可以通过群集中的服务器获得请求的简单分布。

最后，也有可能不使用任何类型的中间媒介，而只是给每个 Web 服务器提供相同的 IP 地址。在这种情况下，我们确实需要假定服务器全都通过单个广播 LAN 连接起来。所发生的事情是：当 HTTP 请求到达时，连接到这个 LAN 的 IP 路由器将简单地把它转发给所有服务器，这些服务器然后将运行相同的分布式算法，确定性地决定其中哪些服务器将处理请求。

在 Cardellini 等人的优秀调查报告中描述了组织 Web 群集的不同方式，以及像我们上面所讨论的替代方案。有兴趣的读者可以参阅他们的论文，以获取更多的详细信息和参考。

12.3 通　　信

当涉及基于 Web 的分布式系统时，将只会使用几种通信协议。首先，对于传统的 Web 系统，HTTP 是用于交换消息的标准协议。其次，当考虑 Web 服务时，SOAP 是用于消息交换的默认方式。在本节中将相当详细地讨论这两种协议。

12.3.1 超文本传输协议

Web中客户和服务器之间的所有通信都基于**超文本传输协议**(Hypertext Transfer Protocol,HTTP)。HTTP是一个相对简单的客户-服务器协议;客户向服务器发送一条请求消息并等待一条响应消息。HTTP的一个重要属性是它是无状态的。也就是说,它没有任何打开连接的概念,也不要求服务器维护有关其客户的信息。(Fielding等1999)中描述了HTTP。

12.3.1.1 HTTP连接

HTTP基于TCP。无论何时客户向服务器发出一个请求,它都会首先建立一个到服务器的TCP连接,然后通过该连接发送它的请求消息。同一连接也被用来接收响应。通过使用TCP作为其底层协议,HTTP无须考虑请求和响应的丢失问题。客户和服务器可能简单地假设它们发送的消息能够到达另一端。如果确实出现了错误,例如,连接断开或发生了超时,则会报告一个错误。不过,一般不会尝试从失败中恢复。

第一个HTTP版本的问题之一是它使用TCP连接的效率很低。每个Web文档都是由位于同一个服务器上的多个不同文件的集合构建而成的。要正确显示一个文档,这些文件也需要传输给客户。从原理上讲,这些文件中的每一个都是一个文档,客户可以向存储它的服务器发出单独的请求来获取它。

在HTTP版本1.0及更早的版本中,对服务器的每个请求都要求建立一个单独的连接,如图12.10(a)所示。服务器做出响应之后,连接再次被断开。这种连接被称为是**非持久的**(nonpersistent)。非持久连接的一个主要缺点是建立TCP连接的开销比较大。因此,把一个包含所有元素的完整文档发送给客户可能需要花费相当多的时间。

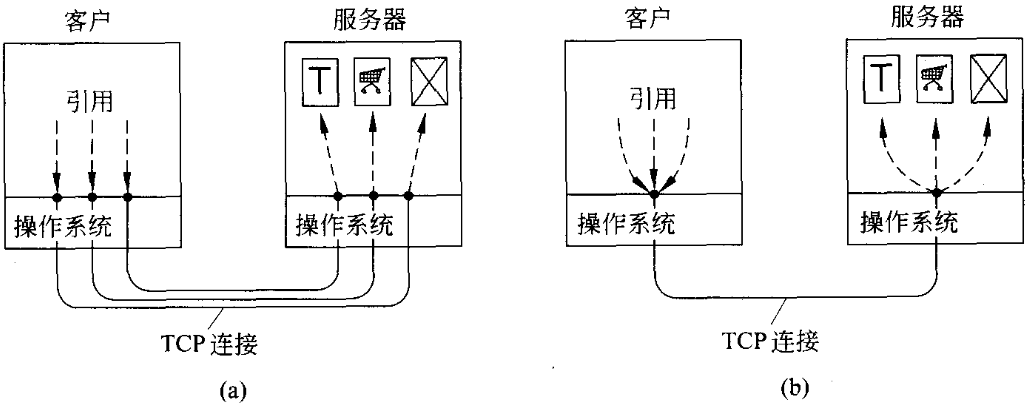


图 12.10
(a) 使用非持久连接;(b) 使用持久连接

注意:HTTP允许一个客户同时和一个服务器建立多个连接。这种方法常常用来抵消建立连接带来的延迟,使数据能够从服务器并行传输给客户。许多浏览器都使用这种方法来改进性能。

HTTP版本1.1采用了另一种方法,它使用**持久连接**(persistent connection),可以在

一个连接中发出多个请求(以及接收它们各自的响应),而无须为每个请求/响应对建立单独的连接。为了进一步改进性能,客户可以连续发出多个请求,而无需等待第一个请求的响应,这也称为**流水线**(pipelining)。图 12.10(b)示意了持久连接的使用。

12.3.1.2 HTTP 方法

HTTP 被设计成一种通用的客户-服务器协议,其设计目标是在这二者之间实现文档的双向传输。客户可以向服务器请求执行这两种操作中的任何一种,方法是把包含所需操作的请求消息发送给服务器。图 12.11 中列出了最常用的请求消息。

操作	说明
head	请求返回一个文档的标题
get	请求把一个文档返回给客户
put	请求存储一个文档
post	提供要添加到一个文档(集合)中的数据
delete	请求删除一个文档

图 12.11 HTTP 支持的操作

HTTP 假设每个文档都有与其相关的元数据,并且文档的元数据存储在一个随请求或响应一起发送的单独的 HTML 文档头部中。当客户不想得到实际的文档而只想得到与其相关的元数据时,会向服务器提交 head 操作。例如,使用该操作可以返回特定文档被修改的时间,这样可以验证由客户缓存的文档的有效性。还可以使用它检查一个文档是否存在,而无需实际地传输文档。

最重要的操作是 get。用这个操作来实际地从服务器上获取一个文档,并把它返回给发出请求的客户。可以指定仅当文档在某个特定时间后被修改过时才返回这个文档。HTTP 还允许文档具有关联的**标签**(tag)(以字符串表示),并且仅当文档的标签与某些特定的标签相匹配时才返回这个文档。

put 操作是 get 操作的逆操作。客户可以请求服务器以某个给定的名称(该名称随请求一起发送)存储一个文档。当然,服务器一般不会盲目地执行 put 操作,它只会接受来自授权客户的请求。我们稍后将讨论如何处理这些安全问题。

post 操作与存储文档有些类似,只不过客户将请求把数据添加到一个文档或文档集合中。一个典型的例子是把一篇文章贴到一个新闻组中。与 put 操作相比,post 的显著特征是:post 操作指出一篇文章应"添加"到哪个文档组中,而且文章随该请求一起被发送。与之对照,put 操作传送的是一个文档和服务器存储该文档所使用的名称。

最后,delete 操作用于请求服务器删除在发送给它的消息中指定的文档。删除操作是否真正发生仍然要取决于各种安全措施。甚至有可能服务器本身也没有删除指定文档的适当权限。毕竟,服务器只是一个用户进程。

12.3.1.3 HTTP 消息

客户和服务器之间的所有通信都是通过消息完成的。HTTP 只识别请求和响应消息。一个请求消息由三部分组成,如图 12.12(a)所示。**请求行**(request line)是必不可少的,它

标识出客户希望服务器执行的操作，并且指明与该请求关联的文档。使用一个单独的字段来标识客户期望使用的 HTTP 版本。后面我们会解释附加的消息头部。

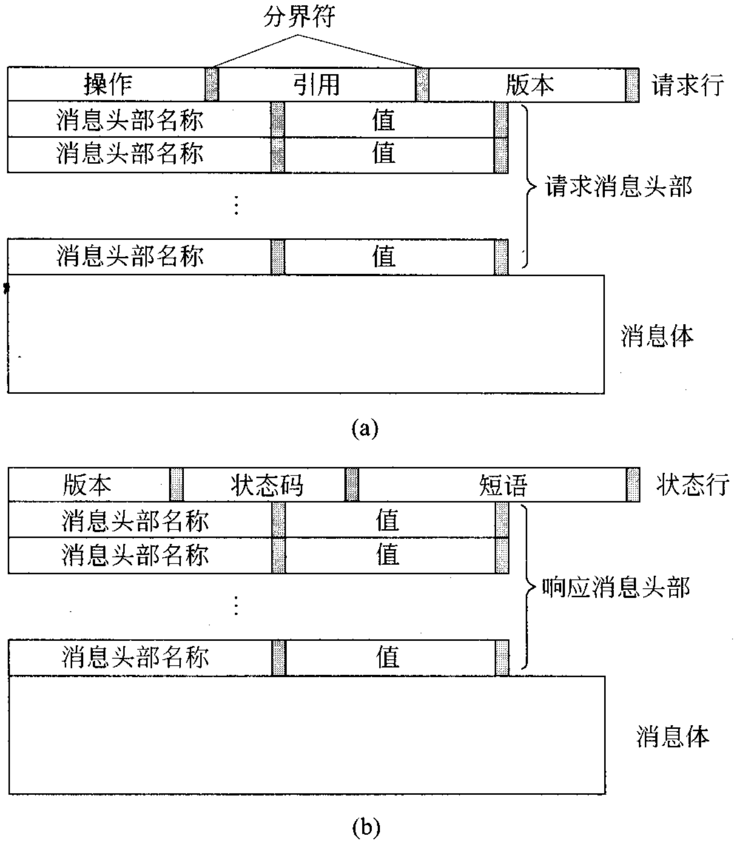


图 12.12

(a) HTTP 请求消息；(b) HTTP 响应消息

响应消息以一个状态行(status line)开始，该行包含一个版本号和一个三位数字的状态码，如图 12.12(b)所示。作为状态行的一部分发送的一个文本短语用来简要地解释状态码的意义。例如，状态码 200 表示一个请求已得到准许，与它相关的文本短语是“OK”。其他常用的状态码还有：

- 400(错误的请求)
- 403(禁止)
- 404(未找到)

请求或响应消息中也许会包含其他头部信息。例如，如果客户对一个只读文档请求了 post 操作，那么服务器的响应消息中会包含状态码 405(“方法不允许使用”)，并且包含一个指明允许的操作(例如，head 和 get)的 Allow 消息头部。另一个例子是，可能仅当一个文档在某个时刻 T 之后被修改过时，客户才对它有兴趣。在这种情况下，客户的 get 请求可以带有指定值 T 的 If-Modified-Since 消息头部作为其参数。

图 12.13 显示了许多可以随请求或响应发送的合法消息头部。大多数头部都是自解释的，因此我们将不会逐一加以介绍。

客户可以向服务器发送不同的消息头部来说明它所能够接受的响应。例如，客户也许可以接受使用在大多数 Windows 和 UNIX 机器上都可用的 gzip 压缩程序压缩的响应信

头部	来源	内容
Accept	客户	客户可以处理的文档类型
Accept-Charset	客户	客户可以接受的字符集
Accept-Encoding	客户	客户可以处理的文档编码
Accept-Language	客户	客户可以处理的自然语言
Authorization	客户	客户证书列表
WWW-Authenticate	服务器	客户应响应的安全口令
Date	客户和服务器	发送消息的日期和时间
ETag	服务器	与返回的文档关联的标签
Expires	服务器	响应保持有效的时间
From	客户	客户的电子邮件地址
Host	客户	文档的服务器的 DNS 名称
If-Match	客户	文档应具有的标签
If-None-Match	客户	文档不应具有的标签
If-Modified-Since	客户	告诉服务器仅当文档在指定时间之后被修改过时才返回该文档
If-Unmodified-Since	客户	告诉服务器仅当文档在指定时间之后未被修改过时才返回该文档
Last-Modified	服务器	返回的文档上次被修改的时间
Location	服务器	客户应把请求重定向到的文档引用
Referer	客户	指向客户最近请求过的文档
Upgrade	客户和服务器	发送方希望切换到的应用程序协议
Warning	客户和服务器	消息中数据状态的信息

图 12.13 一些 HTTP 消息头部

息。这时,客户将随请求发送一个 Accept-Encoding 消息头部,其内容是“Accept-Encoding: gzip”。与此类似,例如,Accept 消息头部可以用来指定只能返回 HTML Web 页面。

有两个消息头部是有关安全方面的,但我们本节稍后会讨论到,Web 安全性通常由一个单独的传输层协议来处理。

Location 和 Referer 消息头部用于把客户重定向(redirect)到另一个文档(注意:在规范中,“Referer”是一个错误的拼写)。重定向相当于使用转发指针(forwarding pointer)来定位文档,如第 5 章中所解释的那样。当客户发出一个对文档 D 的请求时,服务器的响应中也许会含有一个 Location 消息头部,指示客户应该重新请求文档 D′。在使用对 D′的引用时,客户可以添加一个 Referer 消息头部,其中包含对 D 的引用以指示导致重定向的原因。一般来讲,这个消息头部用来表示客户最近请求的文档。

Upgrade 消息头部用于切换到另一种协议。例如,客户和服务器可能在开始时使用普通的 HTTP/1.1 来建立连接。然后服务器可能立即在响应中通知客户它希望使用一个安全的 HTTP 版本继续通信,例如使用 SHTTP(Rescorla 和 Schiffman,1999)。在这种情况下,服务器可以发送一个内容为“Upgrade:SHTTP”的 Upgrade 消息头部。

12.3.2 简单对象访问协议

HTTP 是用于传统的基于 Web 的分布式系统的标准通信协议，而**简单对象访问协议**(Simple Object Access Protocol，SOAP)则构成了与 Web 服务进行通信的标准(Gudgin 等 2003)。SOAP 甚至使得 HTTP 比以往更重要：大多数 SOAP 通信都是通过 HTTP 实现的。SOAP 本身是一个并不困难的协议。其主要目的是提供相对简单的方式，允许相互可能了解很少的通信双方能够顺利地通信。换句话说，在设计该协议时，就假定通信双方具有极少的共识。

基于这种假定，SOAP 消息在很大程度上基于 XML 应该就不会令人感到奇怪了。回忆可知：XML 是一种元置标语言，这意味着 XML 描述包括用于描述文档的元素的定义。实际上，这意味着用于消息的语法的定义是该消息的一部分。提供这种语法允许接收方解析各种不同类型的消息。当然，消息的含义仍然是未定义的，因此当消息进入时要采取什么动作也是未定义的。如果接收方无法理解消息的内容，将不能继续进行通信。

SOAP 消息一般由两部分组成，它们一起存放在 **SOAP 信封**(SOAP envelope)中。主体包含实际的消息，而头部则是可选的，其中包含与发送方到接收方的路径上的结点相关的信息。通常，这种结点由 Web 服务的多层实现中的多个进程组成。信封中的所有内容(即头部和主体)都是用 XML 表示的。

看起来似乎很奇怪，SOAP 信封不包含收件人的地址。SOAP 代之以显式假定通过用于传输消息的协议来指定收件人。为此，SOAP 指定了对底层传输协议的**绑定**(binding)。目前，存在两种这样的绑定：一种是对 HTTP 的绑定，另一种是对 SMTP(Internet 邮件传输协议)的绑定。因此，例如，当把一条 SOAP 消息绑定到 HTTP 时，将以 URL 的形式指定收件人，而对 SMTP 的绑定则将以电子邮件地址的形式指定收件人。

这两种不同类型的绑定还指示两种风格的交互。第一种(也是最常见的一种)是**会话式交换风格**(conversational exchange style)。在这种风格中，双方实质上交换的是结构化文档。例如，当以电子方式预订航班时，这种文档可能包含人们将要填写的完整订单。对这种订单的响应可能是一个确认文档，现在包含订单号、航班信息、座位预留，也许还包含在登机时需要扫描的条形码。

与之相反，在调用 Web 服务时，**RPC 风格的交换**(RPC-style exchange)更接近于传统的请求-响应行为。在这种情况下，SOAP 消息将显式确定要调用的过程，还会提供参数值列表作为该调用的输入。此外，响应将是一条正式的消息，其中包含对调用的响应。

通常，通过绑定到 HTTP 来支持 RPC 风格的交换，而会话风格的消息将绑定到 SMTP 或 HTTP。不过，实际上大多数 SOAP 消息都是通过 HTTP 发送的。

一个重要的事实是：尽管由于 XML 中的语法定义现在是消息的一部分而使人们能够更容易地使用常规的解析器，但是 XML 语法本身极其冗长。因此，解析 XML 消息在实际中通常会引入一个严重的性能瓶颈(Allman 2003)。从这方面讲，人们在改进 XML 性能上投入的注意力相对较少就有点令人奇怪了，尽管正在开发一些解决方案(Kostoulas 等 2006)。

同样令人奇怪的是，许多人相信人们可以方便地阅读 XML 规范。图 12.14 中所示的

示例取自官方SOAP规范(Gudgin等2003)。要发现这个SOAP消息传达的是什么内容，将需要进行一些探索，并且不难想象这种晦涩可能是使用XML的自然的副产品。这样，就会想到一个问题，在XML之后基于文本的方法是否是合适的方法：没有人可以方便地阅读XML文档，并且解析器的运行速度非常缓慢。

```
<env:Envelop xmlns:env="http://www.w3.org/2003/05/soap-envelope">
  <env:Header>
    <n:alertcontrol xmlns:n="http://example.org/alertcontrol">
      <n:priority>1</n:priority>
      <n:expires>2001-06-22T14:00:00-05:00</n:expires>
    </n:alertcontrol>
  </env:Header>
  <env:Body>
    <m:alert xmlns:m="http://example.org/alert">
      <m:msg>Pick up Mary at school at 2pm</m:msg>
    </m:alert>
  </env:Body>
</env:Evelope>
```

图12.14 基于XML的SOAP消息的示例

12.4 命　　名

Web使用单一命名系统来引用文档，使用的名称被称为**统一资源标识符**(Uniform Resource Identifiers,URI)(Berners-Lee等2005)。URI有两种形式。一种URI是**统一资源定位符**(Uniform Resource Locator,URL)，它通过包含如何及在哪里访问文档的信息来标识一个文档。换句话说，URL是与位置相关的对文档的引用。相比之下，**统一资源名称**(Uniform Resource Name,URN)更像第5章中讨论的真正的标识符。URN用来作为对文档的全局唯一的、与位置无关的持久引用。

一个URI的实际语法由与其相关的**模式**(scheme)决定。模式名是URI的一部分。已定义的模式有许多种，下面我们将介绍其中的一些模式及其关联的URI。最著名的模式是http，但它不是唯一的模式。我们还应该注意到URL与URN之间的差别正在逐渐减小。作为替代，现在常见的做法是简单地定义URI名称空间(Daigle等2002)。

就URL来说，我们看到它们通常包含关于如何及在哪里访问一个文档的信息。访问文档的方法通常由作为URL一部分的模式名(例如http、ftp或telnet)来反映。文档所在的位置通常嵌入在URL中，由可以向其发送访问请求的服务器的DNS名称指定，尽管也可以使用IP地址。服务器用来侦听访问请求的端口号也是URL的一部分，如果这部分被省略，则使用默认端口。最后，URL还包含服务器查找文档时使用的文档名。URL的一般结构如图12.15所示。

解析图12.15中所示的URL是很简单的。如果使用DNS名称指定服务器，那么需要

模式		主机名	路径名
http	://	www.cs.vu.nl	/home/steen/mbox

(a)

模式		主机名		端口	路径名
http	://	www.cs.vu.nl	:	80	/home/steen/mbox

(b)

模式		主机名		端口	路径名
http	://	130.137.24.11	:	80	/home/steen/mbox

(c)

图 12.15　URL 的常用结构：

(a) 只使用 DNS 名称；(b) 结合使用 DNS 名称和端口号；(c) 结合使用 IP 地址和端口号

把 DNS 名称解析为服务器的 IP 地址。利用 URL 中包含的端口号，客户可以使用由模式指定的协议来联系服务器，然后向服务器传递 URL 最后一部分中的文档名。

尽管 URL 在 Web 中仍然很常见，但是已经为其他类型的 Web 资源提议了多个不同的 URI 名称空间。图 12.16 中列出的是 URI 的一些例子。如前所述，http 模式的 URI 使用 HTTP 传输文档。与此类似，ftp 模式的 URI 使用 FTP 传输文件。

名称	用于	示例
http	HTTP	http://www.cs.vu.nl:80/globe
mailto	电子邮件	mailto:steen@cs.vu.nl
ftp	FTP	ftp://ftp.cs.vu.nl/pub/minix/README
file	本地文件	file:/edu/book/work/chp/11/11
data	内联数据	data:text/plain;charset=iso-8859-7,%e1%e2%e3
telnet	远程登录	telnet://flits.cs.vu.nl
tel	电话	tel:+31201234567
modem	调制解调器	modem:+31201234567;type=v32

图 12.16　URI 示例

使用 data 模式的 URI(Masinter 1998)支持直接的文档形式。在这样的 URI 中，文档本身嵌入在 URI 中，这类似于把文件的数据嵌入在一个索引结点(inode)中(Mullender 和 Tanenbaum 1984)中。示例 URI 中包含了代表希腊字符串 αβγ 的纯文本。

URI 除了用来指向一个文档外，还常常用于其他用途。例如，telnet URI 可以用来和服务器建立一个 telnet 会话。还有用于基于电话通信的 URI，在(Schulzrinne 2005)中有相关描述。tel URI 如图 12.16 中的例子所示，它实质上只嵌入了一个电话号码，并简单地让客户通过电话网建立一次通话。在这种情况下，客户通常是电话设备。modem URI 可以用来与另一台计算机建立基于调制解调器的连接。示例 URI 指定远程调制解调器应该遵守 ITU-T V32 标准。

12.5 同　步

对于大多数传统的基于 Web 的系统而言，同步并不算是一个大问题，这主要出于两个原因。第一，Web 采用严格的客户-服务器结构，服务器之间（或客户之间）从不相互交换信息，所以不会出现同步问题。第二，可以把 Web 看作是一个大多数时候进行读操作的系统。更新操作一般由单个人员或实体完成，因此很少会有写-写冲突发生。

不过，这种情况正在发生变化。例如，越来越多的应用要求对 Web 文档的协同创作提供支持。也就是说，Web 应当为由一组协同工作的用户或进程对文档的并发更新提供支持。此外，随着 Web 服务的引入，需要服务器相互之间进行同步并协调它们的动作。我们在前面已经讨论了 Web 服务中的同步。因此，我们将只简单地关注对 Web 文档的协作维护进行同步。

Web 文档的分布式创作是通过单独的协议（即 WebDAV）处理的（Goland 等 1999）。WebDAV 代表 **Web 分布式创作和版本控制**（Web distributed authoring and versioning），它提供一种简单的方式来锁定共享文档，可以在远程 Web 服务器上创建、删除、复制和移动文档。下面我们简要介绍 WebDAV 中支持的同步。在（Kim 等 2004）中提供了如何在实际设置中使用 WebDAV 的概述。

为了同步对一个共享文档的并发访问，WebDAV 支持一种简单的锁定机制。有两类写入锁。可以把独占式写入锁赋予一个客户，它将阻止任何其他客户在共享文档被锁定时修改该文档。另一类是共享式写入锁，它允许多个客户同时更新文档。因为锁定发生在整个文档的粒度上，所以共享式写入锁对于多个客户修改同一文档的不同部分的情况很方便。不过，此时需要客户自己小心避免发生写-写冲突。

通过向客户传递一个锁令牌来赋予客户一个锁。服务器中注册了哪个客户当前具有锁令牌。无论何时客户希望修改文档，它都会向服务器发送一个 HTTP post 请求，并随请求发送锁令牌。这个令牌表明该客户可以对文档进行写访问，因而服务器将执行该请求。

一个重要的设计问题是：在客户持有锁令牌期间无须在客户和服务器之间维护一个连接。客户在获取锁令牌之后可以简单地断开与服务器之间的连接，并在发送 HTTP 请求时重新连接服务器。

注意：当一个持有锁令牌的客户崩溃时，服务器必须能以某种方式收回锁。WebDAV 没有指定服务器如何处理这类问题，而是把它留给了具体的实现。其原因是最佳解决方法因 WebDAV 所应用的文档类型而异。采用这种方法的原因是：没有圆满地解决孤儿锁问题的通用方法。

12.6 一致性与复制

在基于 Web 的分布式系统中，最重要的面向系统开发之一也许是：确保对 Web 文档的访问满足严格的性能和可用性要求。这些要求导致了对于缓存和复制 Web 内容的众多

提议，本节中将讨论其中多种提议。原始模式（目前仍然被大量部署）旨在支持静态内容，现在也投入了大量的努力来支持动态内容，即支持作为请求的结果而生成的文档以及那些包含脚本的文档，等等。（Rabinovich 和 Spatscheck 2002）中提供了关于 Web 缓存和复制的优秀、完整的描述。

12.6.1 Web 代理缓存

客户端缓存一般有两种形式。首先，大多数浏览器都带有简单的缓存功能。无论何时获取了一个文档，它都会被存储在浏览器的缓存中，以便下一次从中加载它。客户一般可以通过指定何时进行一致性检查来对缓存进行配置，下面论述一般情况时将对此给出解释。

其次，客户站点通常运行一个 Web 代理。如前所述，Web 代理接受本地客户的请求并将其传递给 Web 服务器。响应到来后，将结果传递给客户。这种方法的优点是代理可以对结果进行缓存，并且在需要时可以把该结果返回给另一个客户。换句话说，Web 代理可以实现共享缓存。

除了在浏览器和代理处进行缓存之外，还可以设置覆盖一个地区甚至一个国家的缓存，这就导致了**分层式缓存**（hierarchical cache）。这种模式主要用于减少网络通信量，但缺点是其延迟时间长于非分层式缓存模式的延迟时间。造成延迟较长的原因是：客户需要检查多个缓存，而不只是非分层式缓存模式中的一个缓存。不过，这种较长的延迟与文档的普及性密切相关：与不流行的文档相比，对于流行的文档来说，在离客户更近的缓存中找到它的一个副本的机会更高。

作为构建分层式缓存的一种替代方法，人们也可以为协作式部署组织缓存，如图 12.17 所示。在**协作式缓存**（cooperative caching）或**分布式缓存**（distributed caching）中，只要 Web 代理的缓存未能命中，该代理首先将检查它邻近的代理中是否包含被请求的文档。如果检查失败，该代理再把请求转发给负责该文档的 Web 服务器，这种模式主要是与 Web 缓存一起部署的，该 Web 缓存属于共存于同一个 LAN 中的相同组织或机构。（Wolman 等 1999）的一项研究表明协作式缓存对于较小的客户团体（数万个用户）可能是有效的，注意到这一点是有趣的。不过，这样的团体也可能会使用单个代理缓存，因为这样在通信和资源使用上的成本更低廉。

（Rodriguez 等 2001）对分层式缓存和协作式缓存所做的比较清楚指出需要进行多种折衷。例如，由于协作式缓存一般是通过高速链路连接的，因此获取一个文档所需的传输时间比分层式缓存要少得多。此外，正如人们所期待的，协作式缓存对存储空间的要求也要比分层式缓存宽松一些。他们还发现，分层式缓存预期的延迟比分布式缓存要短一些。

Web 中部署了不同的缓存一致性协议。为了保证从缓存返回的文档是一致的，一些 Web 代理会先向服务器发送一个有条件的 HTTP get 请求，其中附带的 If-Modified-Since 请求消息头部中指定缓存中的文档的最后一次修改时间。仅当文档在那个时间之后被修改过时，服务器才会返回整个文档。否则，Web 代理只是简单地将其缓存中的文档版本返回给发出请求的本地客户。根据第 7 章中介绍的术语，这对应于一个基于拉的协议。

不幸的是，这种策略要求代理为每一个请求联系服务器。为了用较弱的一致性换取性能的改进，广为采用的 Squid Web 代理（Wessels 2004）根据文档被缓存的时间距离它最后

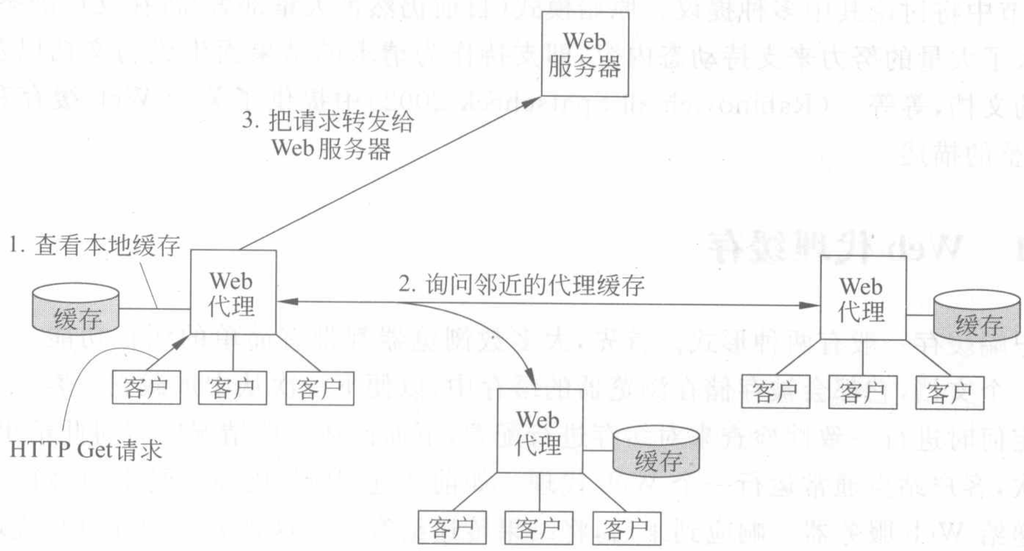


图 12.17　协作式缓存的原理

一次被修改的时间有多久来决定文档的到期时间 T_{expire}。具体地说,如果 $T_{last_modified}$ 是文档最后被修改的时间(由其所有者进行记录),并且 T_{cached} 是它被缓存的时间,那么:

$$T_{expire} = \alpha(T_{cached} - T_{last_modified}) + T_{cached}$$

其中 $\alpha=0.2$(这个值是根据实践经验得出的)。在时间 T_{expire} 之前,认为文档有效,代理不会与服务器联系。在时间到期之后,代理请求服务器发送一个最新的副本,除非文档没有被修改过。也就是说,当 $\alpha=0$ 时,这个策略就是我们刚刚讨论过的那种方法。

请注意,有很长时间未被修改的文档不会像不久前刚修改过的文档那样快地被检查。它的明显缺点是代理可能会返回一个无效的文档,即一个比服务器中存储的当前版本更旧的文档。更糟糕的是,客户没有办法检测到自己收到的是一个过时的文档。

基于拉的协议的一种替代方法是让服务器通过发送失效信息来通知代理文档已被修改。这种针对 Web 代理的方法的问题是服务器可能需要保持大量代理的记录,因而不可避免地导致可伸缩性问题。不过,通过结合使用租约和失效,(Cao 和 Liu 1998)证明服务器需要维护的状态信息可以被控制在可以接受的限度内。注意:这种状态在很大程度上是由为租约设置的到期时间规定的。然而,用于 Web 代理缓存的失效协议目前还很少付诸应用。

在(Cao 和 Oszu 2002)中可以找到各种 Web 缓存一致性策略的比较。他们得出的结论是:就带宽和感觉到的客户延迟而言,让服务器发送失效消息可能优于其他任何方法,同时还可以使缓存的文档与源服务器上的那些文档保持一致。在电子商务应用中,经常可以发现采用这样的访问模式。

Web 代理缓存的另一个问题是它们只能用于存储静态文档。静态文档不是那种应客户的请求而由 Web 服务器动态生成的文档。来自一个客户的相同请求可能下次会得到不同的响应,从这个意义上说,这些动态生成的文档通常是唯一的。例如,许多文档包含广告(称为**广告条**(banner)),它会针对每一次请求而发生变化。下面当我们讨论 Web 应用程序的缓存和复制时,将回过头来讨论这种情形。

最后,我们还应该提及人们进行了大量的研究以查明最佳的缓存替代策略是什么。目前出现了众多提议,但是总的来说,简单的替代策略(比如收回最近最少使用的对象)就可以

足够好地工作。在(Podling 和 Boszormenyi 2003)中介绍了对一些替代策略的深入调查研究。

12.6.2 Web 宿主系统的复制

随着 Web 作为组织展示它们自身以及与最终用户直接交互的工具而变得越来越重要，我们看到了维护 Web 站点的内容与确保站点可以轻松、持续不断地访问之间的变化。这种区别为**内容分发网络**(content delivery network, CDN)的出现铺平了道路。这些 CDN 底下的基本思想是：它们担当 Web 宿主服务，提供一种基础结构，来分布和复制整个 Internet 上的多个站点的 Web 文档。这个基础结构的规模可能令人印象深刻。例如，在 2006 年时，就有报告称 Akamai 有超过 18 000 个服务器散布在 70 个国家。

完整的 CDN 要求自动分布和复制宿主的文档，从而导致了自管理系统的体系结构，如第 2 章中所讨论的那样。在大多数情况下，大规模的 CDN 是沿着反馈控制环的线路组织的，如图 12.18 所示，在(Sivasubramanian 等 2004b)中详细描述了它。

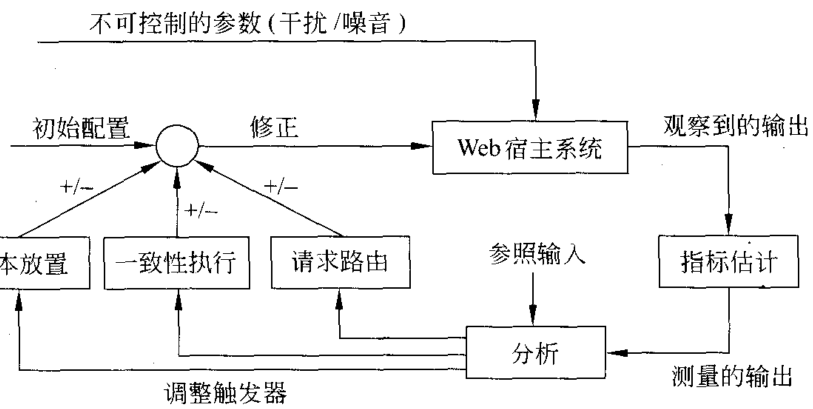


图 12.18 CDN 作为反馈控制系统的一般组织结构(摘自(Sivasubramanian 等 2004b))

实质上有三个不同的方面与 Web 宿主系统中的复制相关：指标估计、修正触发和采取合适的措施。最后一个方面可以细分为副本布置决策、一致性执行和客户请求路由。在下面，我们将逐一简要介绍它们。

12.6.2.1 指标估计

CDN 的一个有趣方面是：当涉及宿主复制的内容时，它们需要在许多方面之间进行折衷。例如，如果一个文档被大量复制，该文档的访问次数可能是最佳的，但是与此同时这却招致了财务成本，以及用于分发更新而招致的带宽使用方面的成本。总的来说，有许多提议用于估计 CDN 工作得有多好。这些提议被分成几个类别。

首先，有**延迟指标**(latency metrics)，通过它来度量执行一个动作(例如，获取一个文档)的时间。这看起来似乎是微不足道的，例如，当决定副本布置的进程需要知道客户与某个远程服务器之间的延迟时，估计延迟时间将变得很困难。通常，将需要部署在第 6 章中讨论过的用于全局定位结点的算法。

除了估计延迟之外，度量两个结点之间的可用带宽可能更重要。当需要传输大型文档

时，这种信息特别重要，就像系统的响应速度在很大程度上是由可以传输文档的时间规定的那样。有多种工具可用于度量可用带宽，但是在所有情况下都证明很难获得准确的度量。在(Strauss 等 2003)中可以找到进一步的信息。

另一个类别由**空间指标**(spatial metrics)组成，它主要包括：依据网络级路由跳段数或者自主系统之间的跳段数来度量结点之间的距离。同样，确定任意两个结点之间的跳段数可能非常困难，并且甚至还可能与延迟不相关(Huffaker 等 2002)。此外，当部署低级技术(如**多协议标签交换**(multi-protocol label switching，MPLS))时，仅仅查看路由表将不起作用。MPLS 通过使用虚电路技术即时、高效地把数据包转发到它们的目的地，避开了网络级路由(Guichard 等 2005)。因此，数据包遵循的路由可能完全不同于网络级路由表中公布的路由。

第三个类别是由**网络使用指标**(network usage metrics)构成的，它必然伴随有消耗的带宽。根据要传输的字节数计算消耗的带宽通常很容易。不过，为了正确地执行该任务，我们需要考虑多久阅读一次文档、多久更新一次文档，以及多久复制一次文档。我们把它作为练习留给读者完成。

一致性指标(consistency metrics)告诉我们一个副本在多大程度上偏离了它的主副本。在第7章中的连续一致性的环境中，我们已经详细讨论了如何度量一致性(Yu 和 Vahdat 2002)。

最后，**财务指标**(financial metrics)构成了用于度量 CDN 工作得有多好的另一个类别。尽管这无关技术，考虑到大多数 CDN 都运行在商业基础上，显然在许多情况下财务指标将是决定性的。此外，财务指标与 Internet 的实际基础结构密切相关。例如，大多数商业 CDN 把服务器放在 Internet 的边缘，这意味着它们从直接为最终用户服务的 ISP 租用容量。此时，业务模型开始与科学技术问题纠缠在一起，而这个领域根本没有得到很好的理解。关于财务特性与科学技术问题之间的关系只有很少的资料可查(Janiga 等 2001)。

从这些例子应该可以清楚看出：只度量 CDN 的性能或者甚至只估计其性能就可能是一项极其复杂的任务。在实际中，对于商业 CDN 来说，确实有价值的问题是：它们是否可以满足与顾客达成的服务级别协议。这些协议通常是简单地依据将会多快地给顾客提供服务来制定的。然后将取决于 CDN 来确保这些协议得到满足。

12.6.2.2 修正触发

需要解决的另一个问题是：何时以及如何触发修正。一个简单的模型是定期估计指标，随后根据需要采取措施。在实际中经常可以看到这种方法。位于服务器上的特殊进程将会收集信息，并定期检查发生的变化。

定期评估的主要缺点是：可能遗漏突然的变化。人们现在相当关注的一类突然的变化是大规模突发访问。**大规模突发访问**(flash crowd)是指突然出现对特定 Web 文档的大量请求。在许多情况下，这类突发事件可能使整个服务崩溃，继而引发一连串的服务停用，因特网的最近历史中发生的几个事件都证明了这一点。

处理大规模突发访问是困难的。一个代价高昂的解决方案是：大量复制一个 Web 站点，一旦请求比率开始快速增加，就应该把请求重定向到副本，以移除主副本上面的负载。这种过度供应方案显然不是我们要走的道路。作为替代，我们需要使用一种**大规模突发访

问预测器(flash-crowd predictor),它将给服务器提供足够的时间来动态安装 Web 文档的副本,然后当进入的请求变得过多时,它就可以重定向它们。尝试预测大规模突发访问的问题之一是:可能极难预测它们。图 12.19 显示了遭受大规模突发访问的 4 个不同的 Web 站点的访问迹线。作为参照点,图 12.19(a)显示了跨越两天的规则访问迹线。还有一些非常强烈的峰值,但是不会令人震惊。与之相比,图 12.19(b)显示了具有 4 次大规模突发访问的两天的迹线。其中仍然有某种规律性,一段时间后就可以发现它,从而可以采取相应的措施。不过,在到达那个时刻之前可能已经造成了损害。

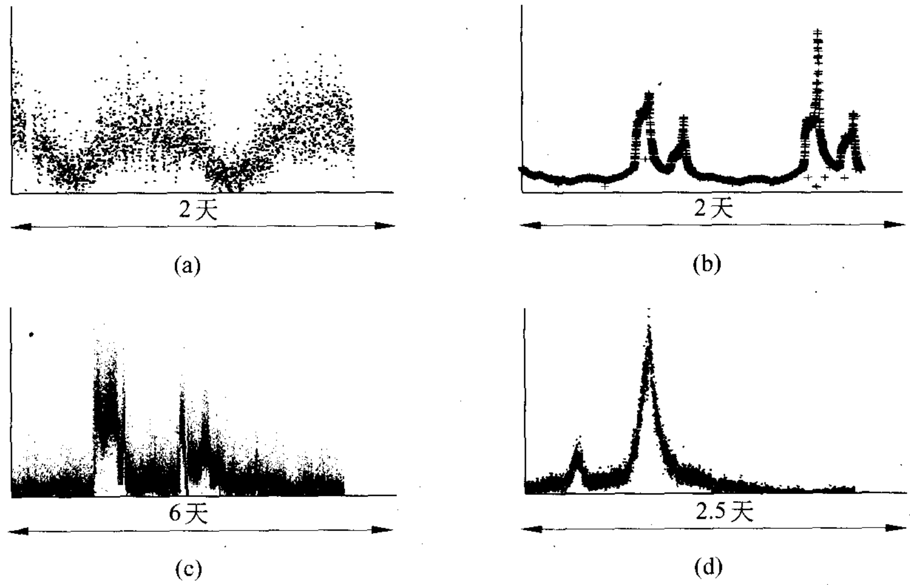


图 12.19 一个正常和三个不同的访问模式,反映了大规模突发访问行为(摘自(Baryshnikov 等 2005))

图 12.19(c)显示了一条跨越 6 天的迹线,其中至少具有两次大规模突发访问。在这种情况下,任何预测器都将具有一个严重的问题,因为它证明这两次请求比较几乎是同时增加的。最后,图 12.19(d)显示了如下情形:第一个峰值可能不会引发任何修正,但是第二个峰值显然应该会这样。事实证明:在这种情形下,可以通过运行时分析相当好地处理其行为类型。

预测大规模突发访问的一种有前途的方法是:使用简单的线性外推法技术。(Baryshnikov 等 2005)提议在特定的时间间隔[t-W,t)内持续不断地测量请求数并将其记入一个文档中,其中 W 是**窗口大小**(window size)。把时间间隔本身分成较小的时隙,并且统计每个时隙的请求数。然后,通过应用简单的线性回归,我们可以拟合一条曲线 f_t,把访问次数表示成时间的函数。通过把该曲线外推到 t 以外的时间实例,我们就可以获得请求次数的预测。如果预测请求次数超过了给定的阈值,就会引发警报。

这种方法对于多种访问模式工作得相当好。不幸的是,窗口大小以及确定警报阈值是什么被认为高度依赖于 Web 服务器通信量。在实际中,这意味着将需要进行大量的手工微调,以便为特定的站点配置理想的预测器。如何自动配置大规模突发访问预测器目前尚不为人所知。

12.6.2.3 调整措施

如前所述,实质上只能采取三种(相关的)措施来改变 Web 宿主服务的行为:改变副本

的布置、改变一致性执行策略、决定如何以及何时重定向客户请求。我们已经在第7章中详细讨论了前两种措施。客户请求重定向值得引起更多的注意。在讨论一些折衷方法之前，让我们首先通过考虑 Akamai 情形（Leighton 和 Lewin 2000；Dilley 等 2002），看看在实际设置中如何处理一致性和复制。

其基本思想是：每个 Web 文档都包含一个主 HTML（或 XML）页面，其中嵌入了多个其他的文档，如图像、视频和音频。为了显示完整的文档，就必须让用户的浏览器也获取嵌入式文档。假定这些嵌入式文档很少发生变化，这样缓存或复制它们就有意义了。

通常通过一个 URL 来引用每个嵌入式文档。不过，在 Akamai 的 CDN 中，把这样的 URL 修改成引用**虚拟 ghost**（virtual ghost），它是指向 CDN 中的实际服务器的引用。该 URL 还包含源服务器的主机名称，我们下面将解释其原因。像下面这样解析修改过的 URL，也如图 12.20 所示。

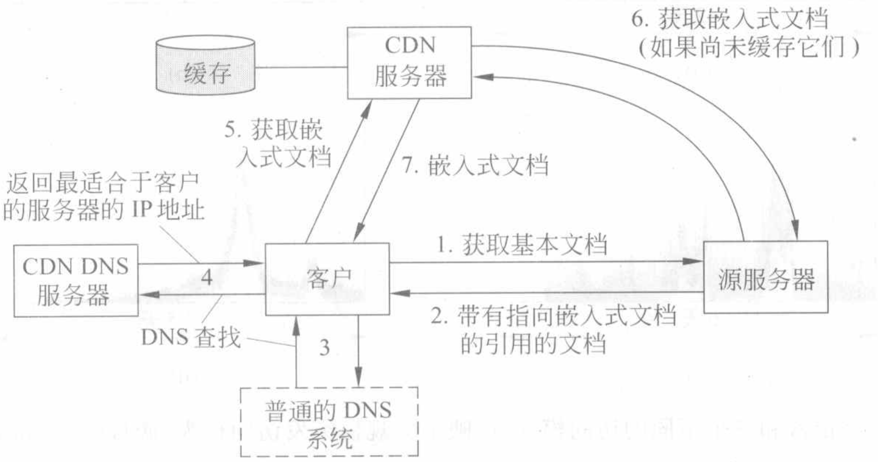


图 12.20　Akamai CDN 的基本工作原理

虚拟 ghost 的名称中包括一个 DNS 名称（比如 ghosting.com），它被普通的 DNS 命名系统解析到一个 CDN DNS 服务器（第3步的结果）。每个这样的 DNS 服务器都会记录靠近客户的服务器。此时，可以使用我们以前讨论过的任何接近度指标。实际上，CDN DNS 服务器将把客户重定向到最适合于该客户的副本服务器（第4步），它可能意指距离最近的服务器、负载最少的服务器，或者其中几个指标的组合（实际的重定向策略是专有的）。

最后，客户把对嵌入式文档的请求转发给所选的 CDN 服务器。如果这个服务器尚不具有文档，它就会从原始 Web 服务器上获取它（如第6步所示），把它缓存在本地，随后把它传递给客户。如果文档已经在 CDN 服务器的缓存中，就可以立即返回它。注意：为了获取嵌入式文档，副本服务器必须能够发送一个请求给源服务器，因此它的主机名称也会包含在嵌入式文档的 URL 中。

这种模式的一个有趣方面是简单性，通过它可以执行文档的一致性策略。显然，无论是否更改主文档，客户总是能够从源服务器上获取它。对于嵌入式文档，将需要对这些文档使用一种不同的方法。在实际中，将从邻近的副本服务器获取文档。此时，嵌入式文档的 URL 不仅会引用最终通向 CDN DNS 服务器的特殊主机名称，而且会包含一个唯一标识符，每当嵌入式文档发生变化时，它也会相应地发生变化。实际上，这个标识符将会改变嵌入式文档的名称。因此，当把客户重定向到特定的 CDN 服务器时，该服务器将不会在其缓

存中找到指定的文档，因而会从源服务器上获取它。旧文档最终会被逐出服务器的缓存，因为不再会引用它。

这个示例已经说明了客户请求重定向的重要性。从原理上讲，通过正确重定向客户，当涉及客户感知的性能时，CDN 可以保持控制，而且可以通过(例如)避免把请求发送给负载很重的服务器来考虑全局系统性能。当把关于系统的当前行为的信息提供给执行重定向决策的进程时，就可以应用这些所谓的**自适应重定向策略**(adaptive redirection policy)。这把我们部分带回到前面讨论过的指标估计技术上。

除了不同的策略之外，一个重要的问题是请求重定向是否对客户透明。实质上讲，只有三种重定向技术：TCP 移交、DNS 重定向和 HTTP 重定向。我们已经讨论了 TCP 移交。这种技术仅适用于服务器群集，还没有扩展到广域网。

DNS 重定向是一种透明的机制，通过这种机制，客户可以对文档所在的位置一无所知。Akamai 的两级重定向是这种技术的一个例子。如我们以前所讨论的，我们也可以直接部署 DNS 来返回多个地址中的一个。不过，注意，DNS 重定向只适用于整个站点：各个文档的名称并不适合于 DNS 名称空间。

最后，HTTP 重定向是一种非透明的机制。当客户请求特定的文档时，可能作为 HTTP 响应消息的一部分给它提供一个替代 URL，然后将把它重定向到这个 URL。一个重要事项是：这个 URL 对于客户的浏览器是可见的。事实上，用户可能决定把推荐的 URL 记入书签，这潜在地会导致重定向策略无用。

12.6.3 Web 应用程序的复制

至此，我们主要关注的是缓存和复制静态 Web 文档。在实际中，我们看到 Web 日益提供更多动态生成的内容，但是它也扩展到提供可以被远程应用程序调用的服务。同样在这些情形下，我们看到缓存和复制有助于显著改进整体性能，尽管到达这种改进的方法比我们迄今为止所讨论的要精巧一些(Conti 等 2005)。

在考虑通过缓存和复制改进 Web 应用程序的性能时，如果可以部署多个解决方案，但是没有最一个被证明是最佳的，那么就会使事情变得复杂。让我们考虑如图 12.21 所示的边缘服务器情形。在这种情况下，我们假定有一个 CDN，其中的每个托管站点都有一个源服务器充当所有读取和更新操作的授权站点。边缘服务器用于处理客户请求，并且能够存储与源服务器上保存的(部分)相同的信息。

回忆可知：在边缘服务器体系结构中，Web 客户通过边缘服务器请求数据，该边缘服务器反过来又从与客户查阅的特定 Web 站点关联的源服务器获取它的信息。同样如图 12.21 所示，我们假定源服务器包含一个从其中动态创建响应的数据库。尽管我们只显示了一个 Web 服务器，但是常见的做法是依据我们以前讨论过的多层体系结构来组织每个服务器。现在可以根据下面的介绍粗略地组织边缘服务器。

首先，为了改进性能，我们可以决定应用存储在源服务器上的数据的完全复制。无论何时更新比率较低并且当查询需要进行代价高昂的数据库搜索时，这种模式将工作得很好。如前所述，我们假定所有的更新都是在源服务器上执行的，该服务器负责使副本和边缘服务器保持一种一致的状态。因此，读取操作可以发生在边缘服务器上。我们在这里看到：当

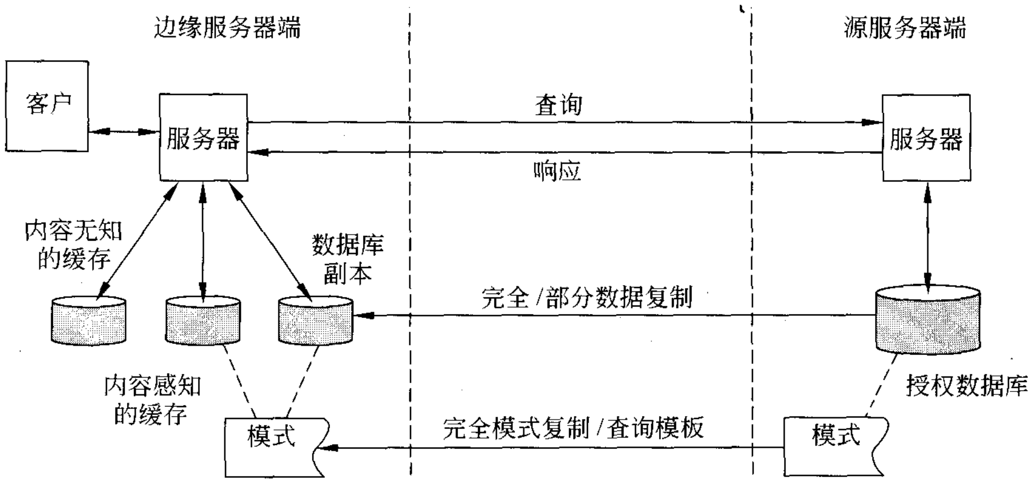


图 12.21　利用 Web 应用程序进行缓存和复制的替代方案

更新比率较高时，为改进性能而进行的复制将失败，因为每个更新操作都会引发经由广域网的通信，以使副本保持一致的状态。如(Sivasubramanian 等 2004a)中所说明的那样，读取/更新比率是在广域设置中应该把源数据库复制到何种程序的决定性因素。

当查询一般很复杂时，也需要进行完全复制。就关系数据库来说，这意味着查询需要搜索和处理多个表，在执行连接操作时一般也是这样。与复杂查询相对的是简单查询，它一般只需要访问一个表以产生一条响应。对于后一种情况，**部分复制**(partial replication)可能就足够了，这样只会把数据的一个子集存储在边缘服务器上。

部分复制的问题是：可以难以手动决定在边缘服务器上需要哪些数据。(Sivasubramanian 等 2005)提议自动处理这种情况，其方法是根据 Globule 复制其 Web 页面的相同原理来复制记录。如我们在第 2 章中所讨论的，这意味着源服务器分析数据记录的访问迹线，然后依据它决定把记录放在什么位置。回想一下在 Globule 中，一旦数据处于合适的位置(并且可能被复制)，考虑执行读取和更新操作的代价来驱动决策。可以用一个简单的线性函数来表示代价：

$$cost = (w_1 \times m_1) + (w_2 \times m_2) + \cdots + (w_n \times m_n)$$

其中 m_k 是一个性能指标(比如消耗的带宽)；$w_k > 0$，它是一个相对加权，指示该指标有多重要。

部分复制的一种替代方法是利用**内容感知的缓存**(content-aware cache)。这种情况下的基本思想是：边缘服务器维护一个本地数据库，它现在被定制成可以在源服务器上处理的查询类型。为了对此进行解释，考虑在一个成熟的数据库系统中，将对一个数据库执行查询，该数据库中的数据被组织进表中，以便(例如)把冗余性降至最低。这样的数据库也被称为**规范化**(normalized)。

在这种数据库中，从原理上讲，可以处理遵守数据模式的任何查询，尽管代价也许相当高。利用内容感知的缓存，边缘服务器可以维护一个根据查询的结构组织的数据库。这意味着假定查询遵守有限数量的模板，这实际上意味着可以处理的不同查询类型是受限的。在这些情况下，无论何时收到一个查询，边缘服务器都会对照可用的模板来匹配查询，随后查看其本地数据库，如果可能的话就创建一条响应。如果请求的数据不可用，就把查询转发给源服务器，然后缓存响应，再把它返回给客户。

实际上,边缘服务器所做的是检查是否可以利用本地存储的数据来回答一个查询。这也被称为**查询包含检查**(query containment check)。注意:这种数据被存储在本地,作为对以前发出的查询的响应。当倾向于发出重复的查询时,这种方法工作得最好。

内容感知的缓存比较复杂,这部分是由于边缘服务器上的数据需要保持一致的事实。此时,源服务器需要知道哪些记录与哪些模板相关联,以便可以通过(例如)发送一条验证消息给合适的边缘服务器,正确地处理对记录或表的任何更新。内容感知的缓存之所以比较复杂的另一个原因是:在边缘服务器上仍然需要处理查询。换句话说,需要有不可忽略的计算能力来处理查询。考虑到数据库通常构成Web服务器中的性能瓶颈,可能需要替代解决方案。最后,缓存跨多个表的查询的结果以便可以执行查询包含检查实际上并不是无足轻重的。其原因是:结果的组织方式可能与执行查询的表的组织方式差别很大。

这些事实带给了我们第三种解决方案,即**内容无知的缓存**(content-blind caching),在(Sivasubramanian等2006)中详细描述了它。内容无知的缓存的思想很简单:当客户提交一个查询给边缘服务器时,服务器首先为那个查询计算一个唯一的散列值。使用这个散列值,它随后查看其缓存,检查它以前是否处理过这个查询。如果没有,就把查询转发给源服务器,并缓存结果,然后把它返回给客户。如果以前处理过查询,就把以前缓存的结果返回给客户。

这种模式的主要优点是:与以前描述的数据库方法相比,它减少了边缘服务器所需的计算工作量。不过,就存储空间来说,内容无知的缓存可能是一种浪费,因为与内容感知的缓存或数据库复制相比,内容无知的缓存可能包含太多的冗余数据。注意:这种冗余性也可能使保持缓存最新的过程变得复杂,因为源服务器可能需要保持一份关于哪些更新可能潜在地影响缓存的查询结果的准确说明。如前所述,如果假定查询只能匹配一组有限的预定义模板,就可以缓解这些问题。

显然,这些技术同样可以很好地适用于即将出现的Web服务,但是在可以确定稳定的解决方案之前,仍然需要做大量的研究工作。

12.7 容 错 性

基于Web的分布式系统中的容错性主要通过客户端缓存和服务器复制来实现。例如,在HTTP中没有纳入任何特殊的方法来促进容错性和恢复。不过,注意:Web中的高度可用性是通过冗余性来实现的,它在至关重要的服务(如DNS)中利用了普遍可用的技术。举我们以前提过的一个例子,DNS允许返回多个地址作为名称查找的结果。在传统的基于Web的系统中,考虑到服务器的无状态设计以及所提供的内容通常是静态的,可以相对容易地实现容错性。

当涉及Web服务时,具有类似的情况:几乎没有引入任何新技术或特殊技术来处理容错(Birman 2005)。不过,应该清楚的是,屏蔽故障和恢复的问题可能更严重。例如,Web服务支持广域分布式事务,并且解决方案必须处理失败的参与服务或不可靠的通信。

甚至更重要的是,就Web服务而言,我们可能轻松地处理复杂的调用图。注意:在许多基于Web的系统中,计算遵循一个简单的两层客户-服务器调用惯例。这意味着客户调

用服务器，服务器然后计算一个响应，而无需额外的外部服务。也就是说，通常可以通过简单地复制服务器或者部分地依靠结果缓存来实现容错性。

对于 Web 服务不再是这样。在许多情况下，在我们现在处理的多层解决方案中服务器也充当客户。对服务器应用复制意味着调用者和被调用者需要处理复制的调用，就像我们在第 10 章中讨论的复制对象一样。

由于服务被设计成处理 Byzantine 故障而加剧了问题。组件复制在这里扮演着至关重要的角色，但是客户执行的协议也是如此。此外，我们现在不得不面对如下情形：**Byzantine 容错**(Byzantine fault-tolerant，BFT)服务可能需要充当另一个非复制服务的客户。在(Merideth 等 2005)中提出了这个问题的一种解决方案，它基于(Castro 和 Liskov 2002)提出的 BFT 系统，我们在第 11 章中讨论了它。

有三个问题需要处理。第一，BFT 服务的客户应该只把该服务看作另一种 Web 服务。特别是，这意味着该服务的内部复制以及响应的正确处理对于客户应该是隐藏的。例如，客户需要从至多 2k+1 个响应中收集 k+1 个完全相同的应答，假定 BFT 服务被设计成处理最多 k 个失败的进程。通常，可以在客户端存根中隐藏这种响应处理，可以从 WSDL 规范中自动生成这些客户端存根。

第二，在充当客户时，BFT 服务应该保证内部一致性。特别是，它需要处理如下情况：它调用的外部服务把不同的应答返回给不同的副本。可能会发生这种情况，例如，当外部服务本身由于任何原因而失败时。因此，副本可能需要运行一个额外的协议，作为对它们已经执行用于提供 Byzantine 容错性的协议的扩展。在执行这个协议后，它们可以把它们的应答发送回客户。

最后，外部服务还应该把充当客户的 BFT 服务视作单个实体。特别是，服务不能简单地接受来自单个副本的请求，而是仅当它已经从不同的副本收到至少 k+1 个完全相同的请求时，才可以处理这种请求。

这三种情形导致了三种不同的软件，需要把它们集成进用于开发 Web 服务的工具包中。在(Merideth 等 2005)中可以找到相关的详细信息和性能评估方法。

12.8 安 全 性

考虑到因特网的开放特性，设计一个保护客户和服务器免遭各种攻击的安全体系结构是非常重要的。Web 中的大部分安全问题与建立客户和服务器之间的安全信道有关。在 Web 中建立一条安全信道的最主要方法是使用**安全套接字层**(Secure Socket Layer，SSL)，该协议最初由 Netscape 提出。尽管 SSL 从未被正式标准化，但大多数 Web 客户和服务器都支持它。在 RFC 2246 和 RFC 3546 中正式规定了 SSL 的一个更新，称之为**传输层安全**(Transport Layer Security，TLS)协议(Dierks 和 Alien 1996；Blake-Wilson 等 2003)。

如图 12.22 所示，TLS 是一个与应用程序无关的安全协议，在逻辑上它位于传输层协议之上。为了简单起见，TLS(和 SSL)的实现通常基于 TCP。TLS 可以支持多种高层协议，其中包括 HTTP，我们下面还会讨论这一点。例如，可以使用 TLS 实现 FTP 或 Telnet 的安全版本。

TLS本身被组织为两层。协议的核心由**TLS记录协议层**(TLS record protocol layer)构成,它实现了客户和服务器之间的安全信道。信道的确切特征在建立信道时确定,但可能会包含消息的分段和压缩,以保证消息的身份认证、完整性和机密性。

建立一条安全信道需要经历两个阶段,如图12.23所示。首先,客户通知服务器自己能处理的加密算法以及它所支持的压缩方法。实际的选择通常由服务器做出,它把这个选择报告给客户。这些就是图12.23中所示的前两条消息。

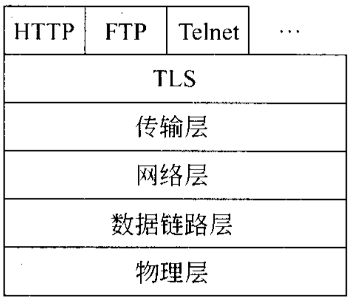


图12.22 TLS在因特网协议栈中的位置

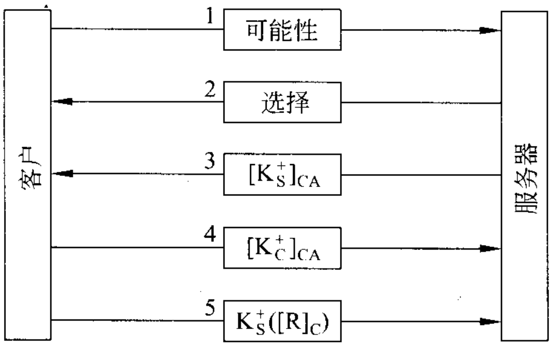


图12.23 带有相互身份认证的TLS

在第二个阶段中进行身份认证。总是要求服务器验证自己的身份,所以它向客户传递一个包含其公钥的证书,该证书由一个证书颁发机构CA签发。如果服务器要求验证客户,客户必须也向服务器发送一个证书,如图12.23中的消息4所示。

客户生成一个可供双方构造会话密钥的随机数,并使用服务器的公钥把这个随机数加密后发送给服务器。另外,如果要求对客户进行身份认证,客户就使用它的私钥对这个随机数进行签名,从而导致图12.23中的消息5(实际上,随单独的消息发送随机数的混合签名版本也可起到同样的作用)。此时,服务器可以验证客户的身份,随后安全信道就建立起来了。

本章小结

可以证明,基于Web的分布式系统使得网络应用程序在最终用户中间很普及。使用Web文档的概念作为交换信息的手段与人们通常在办公室环境和其他场合进行通信的方式很接近。每个人都理解纸张上的文档是什么,因此把这个概念扩展到电子文档对于大多数人来说都是很合乎逻辑的。

提供给Web最终用户的超文本支持对Web的普及起到了极为重要的作用。此外,最终用户一般都会看到一个简单的客户-服务器体系结构,其中可以从特定的站点简单地获取文档。不过,现代的Web站点是依照多层体系结构组织的,其中最后一个组件只负责生成HTML或XML页面,可以把它们作为响应显示给客户。

用应用程序代替最终用户给我们带来了Web服务。从科学技术角度讲,Web服务本身一般并不引人注目,尽管它们仍处于早期阶段。不过,重要的是需要能够让授权的客户发现和访问差别很大的服务。因此,人们付出了极大的努力来标准化服务描述、通信、目录和多种交互方式。同样,每个标准本身并不代表特别新的见解,而只是一个对Web服务的扩展有贡献的标准。

Web 中的进程被定制成处理 HTTP 请求，其中 Apache Web 服务器是一个规范的示例。Apache 被证明是用于处理基于 HTTP 的系统的通用工具，而且还能轻松地扩展到为特定的需求（如复制）提供帮助。

因为 Web 运行在因特网上，所以人们投入了大量的注意力通过缓存和复制改进性能。目前已经为客户端缓存开发了或多或少的标准技术，但是对复制来说，已经取得了相当大的进展。特别是当 Web 应用程序的复制非常重要时，事实证明将需要结合使用多种不同的解决方案来获得最佳的性能。

一般使用标准技术来处理容错性和安全性，这些技术长期应用于许多其他类型的分布式系统。

习 题

1. 电子邮件在 Web 的文档模型中占有什么样的地位？
2. 在许多情况下，Web 站点都被设计成供用户访问。不过，当涉及 Web 服务时，我们看到 Web 站点变得彼此之间相互依赖。考虑图 12.3 中的三层体系结构，你期望在哪里看到出现了依赖性？
3. Web 对文档使用一种基于文件的方法。使用这种方法，客户在打开和显示文档之前首先获取整个文档文件。对于多媒体文件，这种方法会造成什么后果？
4. 可以证明：从科学技术的角度讲 Web 服务没有解决任何新问题。认为 Web 服务很重要的非常有说服力的证据是什么？
5. 与像图 12.9 中所示的那样组织 Web 服务器群集的方式相比，使用第 3 章中讨论的分布式服务器来实现这种群集的主要优点是什么？其明显的缺点是什么？
6. 为什么与非持久连接相比，持久连接通常可以改进性能？
7. SOAP 通常被认为遵守 RPC 语义。确实如此吗？
8. 解释插件、applet、servlet 和 CGI 程序之间的区别。
9. 在 WebDAV 中，为了获取写权限，客户只向服务器出示锁定令牌，这么做足够吗？
10. 服务器可以代替 Web 代理来计算文档的到期时间。这种方法有什么好处？
11. 随着 Web 页面变得高度个性化（因为可以根据需要基于每个客户动态地生成它们），可以证明 Web 缓存全都很快将过时。然而，最近任何时间最有可能不会发生这种情况。请解释为什么。
12. Akamai CDN 采用的是基于拉的分布协议还是基于推的分布协议？
13. 简述一种简单的模式。使用该模式，Akamai CDN 服务器无需检查原始服务器就可以查明一个缓存的嵌入式文档是陈旧的。
14. 与使用一种或几种全局策略相比，为每个 Web 文档分别关联一种复制策略是否有意义？
15. 假设每秒钟对大小为 s 字节的非复制文档请求 r 次。如果把该文档复制到 k 个不同的服务器，并且假定把更新单独传播给每个副本，与不复制文档相比，何时进行复制的代价更低？
16. 考虑一个正在经历大规模突发访问的 Web 站点。为了确保客户仍然会得到良好的服务，可以考虑采取什么合适的措施？

17. 从原理上讲，有三种不同的技术用于把客户重定向到服务器：TCP 移交、基于 DNS 的重定向和基于 HTTP 的重定向。每种技术的主要优缺点各是什么？
18. 给出一个示例，其中由支持内容感知的缓存的边缘服务器执行的查询包含检查将成功返回。
19. （**实验**）建立一个简单的基于 Web 的系统，其方法是：为你的本地机器安装和配置 Apache Web 服务器，使得可以从本地浏览器访问它。如果你在局域网中具有多台计算机，确保可以从该网络上的任何浏览器访问服务器。
20. （**实验**）WebDAV 受到 Apache Web 服务器的支持，并且允许多个用户共享文件，以便通过 Internet 读、写它。在局域网中为支持 WebDAV 的目录安装和配置 Apache。使用 WebDAV 客户测试配置。

第13章

基于协作的分布式系统

在前面几章中我们介绍了分布式系统的一些不同实现方法，每一章都采用一种数据类型作为分布的基础。这些数据类型的例子有对象、文件或(Web)文档，它们都源于非分布式系统。在分布式系统中，对它们进行了修改，以使许多有关分布的问题对用户和开发人员透明。

本章中我们将考虑新一代的分布式系统，它假设系统的各个组件本质上是分布式的，开发这样的系统的真正问题在于对不同组件的活动进行协调。也就是说，不把重点放在组件的透明分布上，而是放在组件间的活动协作上。

本章中介绍的一些有关协作的问题在前面的章节中(特别是在考虑基于事件的系统时)已有所提及。事实证明，许多传统的分布式系统正在逐步结合一些在基于协作的系统中发挥重要作用的机制。

在介绍系统实用示例之前，我们简要介绍一下分布式系统中协作的概念。

13.1 协作模型介绍

基于协作的系统所采用方法的关键之处在于把计算和协作分离开来。如果我们把一个分布式系统视为(可能是多线程)进程的集合，那么分布式系统的计算部分将由下面这样的进程构成：每个进程处理一个特定的计算活动，并且从原理上讲这个活动的执行独立于其他进程的活动。

在这个模型中，分布式系统的协作部分处理进程间的所有通信和协作。它构成把由进程执行的各个活动结合为一个整体的粘合剂(Gelernter 和 Carriero 1992)。在基于协作的分布式系统中，重点放在如何进行进程间的协作上。

(Cabri 等 2000)为可同等应用于许多其他类型分布式系统的移动代理提供了一种协作模型的分类法。为了把他们的术语修改为适用于一般的分布式系统，我们从两个不同的角度(现时性和引用性)对各种协作模型进行分类，如图 13.1 所示。

		现时性：耦合的	现时性：解耦合的
引用性	耦合的	直接	邮箱
	解耦合的	面向会议	生成通信

图 13.1 协作模型的一种分类法
(摘自(Cabri 等 2000))

如果进程之间的耦合是现时性的和引用性的，那么协作就以一种直接的方式进行，这种方式称为**直接协作**(direct coordination)。

引用性耦合一般表现为以显式引用的形式进行通信。例如，一个进程要想和其他进程交换信息，它就必须知道对方进程的名字或标识符才能与之通信。现时性耦合意味着相互通信的两个进程必须都正在运行。这种耦合与我们在第4章中讨论的面向消息的瞬时通信类似。

如果进程之间是现时性解耦合的，但又是引用性耦合的，那么这是另一种不同类型的协作，称为**邮箱协作**(mailbox coordination)。在这种情况下，要进行通信，两个相互通信的进程不必同时运行，而是把消息放到一个(可能是共享的)邮箱中。这种情形类似于我们在第4章中描述的面向消息的持久通信。显式地指明将要存放消息的邮箱是必需的。因此，这是一种引用性耦合。

引用性解耦合系统与现时性耦合系统的结合构成了**面向会议的协作**(meeting-oriented coordination)模型。在引用性解耦合系统中，进程并不显式地知道对方进程。也就是说，当一个进程希望与其他进程协作活动时，它不能直接引用另一个进程，而是使用一个会议的概念，这种概念把进程临时组合在一起以进行协作活动。这种模型决定了参加会议的进程必须同时执行。

基于会议的系统往往通过事件实现，就像基于对象的分布式系统所支持的那样。在本章中，我们讨论实现会议的另一种机制，即基于主题的消息机制。这种机制允许构建称为**发布/订阅系统**(publish/subscribe system)的结构。在这些系统中，一些进程可以订阅包含某个特定主题信息的消息，而其他一些进程则产生(即发布)这些消息。大多数发布/订阅系统要求相互通信的进程同时处于活动状态；因此这是一种现时性耦合。不过，通信进程也可以保持匿名。

最著名的协作模型是引用性解耦合进程和现时性解耦合进程的结合，典型的例子是由Gelernter(1985)在Linda编程系统中引入的**生成通信**(generative communication)。生成通信的关键思想是让一个独立进程的集合共享使用一个持久的元组数据空间。**元组**(tuple)是带有标签的数据记录，它包含若干个(但也可能是零个)各种类型的字段。进程可以把任何类型的记录放入共享的数据空间中(即它们生成通信记录)。与黑板(blackboard)的情况不同，这里无须预先对元组的结构达成一致。只需要使用标签来区分代表不同类型信息的元组。

这些共享的数据空间的一个有趣特性是它们执行一种针对元组的关联搜索机制。也就是说，当一个进程想要从数据空间中提取一个元组时，它实质上是指定自己感兴趣的字段的(一些)值。任何符合该标准的元组都会从数据空间中删除并传递给该进程。如果没有符合条件的元组，进程可以选择被阻塞，直到出现这样的元组为止。后面讨论具体的系统时，我们再介绍这种协作模型的详细信息。

我们注意到生成通信和共享数据空间通常也被认为是一种发布/订阅系统。在下面的内容中，我们也会采用这个共识。在(Eugster等2003)中可以找到发布/订阅系统的良好概述(并且是采用一种相当宽泛的角度介绍它的)。在本章中，我们将采用这些系统中的方法，即进程间至少具有引用性解耦合，不过最好也具有现时性解耦合。

13.2 体系结构

基于协作的系统的一个重要方面是：通过描述将要交换的数据项的特征来进行通信。因此，命名就起到了至关重要的作用。我们将在本章后面介绍命名，但是目前的重要问题是：在许多情况下，发送方和接收方并未显式确定数据项。

13.2.1 一般方法

让我们首先假定数据项是通过一系列**属性**(attribute)描述的。当数据项可供其他进程读取时，就称之为**发布**(published)。此时，需要把**订阅**(subscription)传递给中间件，其中包含订阅者感兴趣的数据项的描述。这样的描述通常包含一些(属性，值)对，可能还结合有(属性，范围)对。在后一种情况下，将期望指定的属性从指定的范围内取值。有时可以使用通过属性表述的各种谓词来给出描述，这种性质非常类似于关系数据库中的类 SQL 查询。我们将在本章后面讨论这类描述。

我们现在面临如下情形：订阅需要**匹配**(matched)数据项，如图 13.2 所示。当匹配成功时，将出现两种可能的情况。在第一种情况下，中间件可能决定把发布的数据转发给它目前的订阅者集合，即具有匹配订阅的进程。另一种选择是，在订阅者可以执行 read 操作来检索发布的数据项那一刻，中间件也可以转发一条**通知**(notification)。

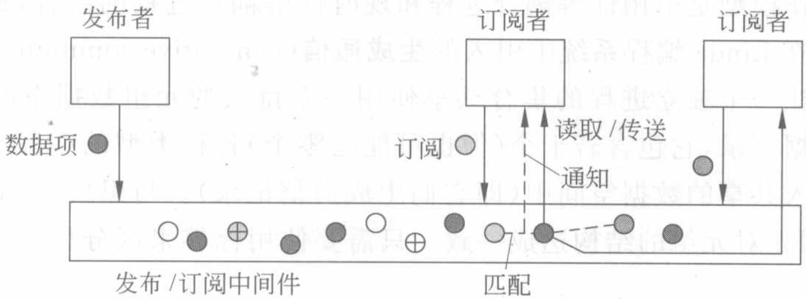


图 13.2　在发布者和订阅者之间交换数据项的原理

在这些情况下，数据项会被立即转发给订阅者，中间件一般不会存储数据。数据存储要么由单独的服务处理，要么由订阅者负责处理。换句话说，我们具有一种引用性解耦合系统，但它不具有现时性解耦合。

当发送通知时，情况将有所不同，这时订阅者需要显式读取发布的数据。此时，中间件必须存储数据项。在这些情况下，有额外的操作用于数据管理。还有可能附加一个租约到数据项上，以便当租约到期时可以自动删除数据项。

在迄今为止描述的模型中，我们假定有 n 个属性 $a_1, \cdots, a_n$ 的固定集合，它们用于描述数据项。特别是，假定每个发布的数据项都具有(属性，值)对一个关联的矢量 $\langle(a_1, v_1), \cdots, (a_n, v_n)\rangle$。在许多基于协作的系统中，这种假设是错误的。而代之以发布**事件**(event)，可以将其视作只具有单个指定属性的数据项。

事件使订阅的处理变得复杂。举例说明，考虑像“当 R4.20 房间空闲并且房门未锁时请发送通知”这样的一个描述。通常，可以实现支持这种描述的分布式系统，其方法是：放置用于监视房间空闲情况的独立传感器(例如，监视传感器)以及用于登记门锁状态的传感器。遵循迄今为止描述的方法，我们将需要把这类原始事件组合到进程可以订阅的可发布的数据项中。事件组合被证明是一项困难的任务，特别是当从分散在整个分布式系统中的不同来源生成这些原始事件时。

显然，在像这样的基于协作的系统中，关键问题是要高效、可伸缩地实现对数据项的匹配订阅，以及构造相关的数据项。从外部讲，由于强大的线程解耦合，协作方法提供了许多潜力用于构建特大型分布式系统。另一方面，如我们下面将看到的，在不损失这种独立性的情况下设计可伸缩的实现是有价值的。

13.2.2　传统的体系结构

用于匹配数据项与订阅的最简单的解决方案是：具有一种集中式客户-服务器体系结构。这是一种目前被许多发布/订阅系统采用的典型解决方案，包括：IBM 的 WebSphere (IBM 2005c)和 Sun 的 JMS 的一些流行的实现(Sun Microsystems 2004a)。此外，更精巧的生成通信模型(如 Jini(Sun Microsystems 2005b)和 JavaSpace(Freeman 等 1999))主要基于中央服务器。让我们探讨两个典型的示例。

13.2.2.1　示例：Jini 和 JavaSpace

Jini 是一个包含各种相关元素的分布式系统。它与 Java 编程语言结合紧密，不过它的许多原理在其他语言中都可以同样实现。该系统的一个重要部分由生成通信的协作模型构成。Jini 通过一个称为 **JavaSpace** 的协作系统(出自于 Linda)来提供进程间的现时性和引用性解耦合(Freeman 等 1999)。一个 JavaSpace 就是一个共享的数据空间，用于存储一些元组，它们代表一组指向 Java 对象的类型化引用。在一个 Jini 系统中可以有多个 JavaSpace 同时存在。

元组以序列化的形式存储。也就是说，每当进程存储一个元组时，首先要对该元组进行编组，这意味着它的所有字段都将被编组。这样，当一个元组包含两个引用同一对象的不同字段时，存储在一个 JavaSpace 实现中的该元组将包含该对象的两个已编组的副本。

通过一个 write 操作可以把元组放入 JavaSpace 中，该操作首先对元组进行编组，然后再存储之。每当对一个元组调用 write 操作时，这个元组的另一个已编组的副本会被存储到 JavaSpace 中，如图 13.3 所示。我们把每个已编组的副本称为一个**元组实例**(tuple instance)。

Jini 中的生成通信的一个有趣方面是从 JavaSpace 中读取元组实例的方式。为了读取一个元组实例，进程会提供另一个元组作为其**模板**(template)，以对 JavaSpace 中存储的元组实例进行匹配。正如任何其他元组一样，模板元组也是对象引用的类型化集合。只会从 JavaSpace 中读出那些与模板元组类型相同的元组实例。模板元组中的一个字段可以包含对一个实际对象的引用，或者包含值 NULL。例如，考虑下面的类：

```
class public Tuple implements Entry {
```

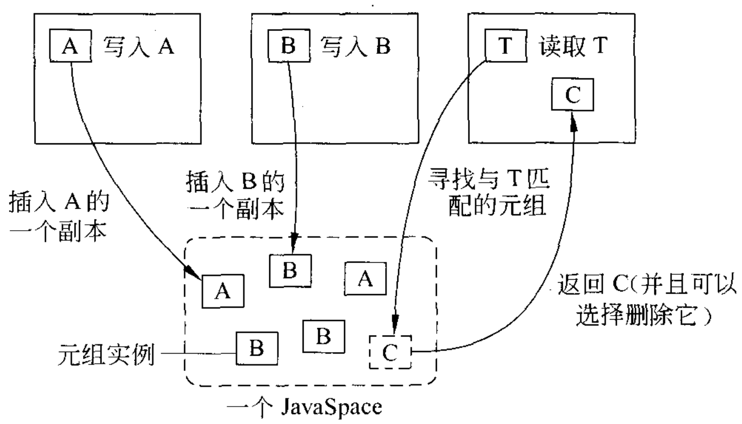


图 13.3 Jini 中的一个 JavaSpace 的一般组织结构

```
    public Integer id, value;
    public Tuple(Integer id, Integer value {this.id = id; this.value = value}
}
```

然后声明一个模板如下：

```
Tuple template = new Tuple(null, new Integer(42))
```

它将匹配下面的元组：

```
Tuple item = new Tuple("MyName", new Integer(42))
```

要把 JavaSpace 中的一个元组实例与一个模板元组相匹配，需要先把模板元组像一般元组那样进行编组(包括它的 NULL 字段)。与模板类型相同的每个元组实例和已编组的模板元组之间会进行逐字段的比较。如果两个字段具有相同的引用副本或者模板元组中的字段为 NULL，那么这两个字段相匹配。如果一个元组实例和模板元组的各个字段分别匹配，那么这个元组实例和模板元组相匹配。

当发现一个元组实例与一个 read 操作提供的模板元组相匹配时，该元组实例将被解编并返回给读取进程。还有一个 take 操作负责从 JavaSpace 中删除元组实例。这两个操作在找到一个匹配的元组实例之前都会阻塞调用程序。可以指定一个最大阻塞时间。另外，在不存在匹配元组时还可以简单地立即返回一些变体。

利用 JavaSpace 的进程不必同时存在。事实上，如果 JavaSpace 是使用持久存储实现的，那么可以停止整个 Jini 系统并在以后重新启动它，而不会丢失任何元组实例。

尽管 Jini 不支持它，也应该清楚具有中央服务器可以允许相当精巧地设计订阅。例如，此刻如果两个非空字段完全相同，那么它们就相匹配。不过，如果认识到每个字段代表一个对象，也可以通过执行对象特有的比较运算符来评估匹配[参见 Picco 等人的著作(2005)]。事实上，如果应用程序可以重写这样的运算符，就可以或多或少地实现任意的比较语义。注意到这种比较可能需要广泛搜索目前存储的数据项是重要的。不能以分布式方式轻松地高效实现这种搜索。由于这种原因，当支持精心设计的匹配规则时，我们一般只会看到集中式实现。

具有集中式实现的另一个优点是：实现同步原语将变得更容易。例如，进程可以阻塞

到直至发布了合适了数据项为止，然后执行一个析构读取操作以删除匹配的元组，这一事实为进程同步提供了便利，而无需进程相互了解对方。同样，如我们在第6章中讨论的那样，分散式系统中的同步天生就比较困难。我们将在下面回过头来讨论同步。

13.2.2.2 示例：TIB/Rendezvous

使用中央服务器的另一种解决方案是：使用多播把已发布的数据项立即传播给合适的订阅者。在TIB/Rendezvous中使用了这种原理，它的基本体系结构如图13.4所示(TIBCO 2005)。在这种方法中，数据项是一条消息，标记有描述其内容的复合关键字，如news.comp.os.books。订阅者提供一个关键字(的某些部分)，或者指示它想接收到的消息，如news.comp.*.books。这些关键字被称为指定消息的**主题**(subject)。

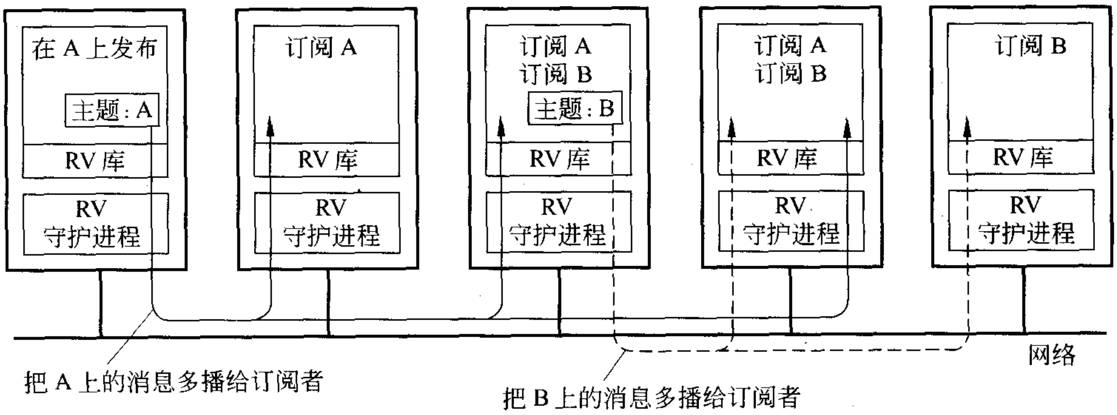


图13.4 TIB/Rendezvous中实现的发布/订阅系统的原理

它的实现的基本原理是使用局域网中常见的广播，不过如果可能，它也使用更高效的通信设施。例如，如果确切地知道一个订阅者所在的位置，通常就会使用点到点的消息。在这样的网络中，每个主机都将运行一个**Rendezvous守护进程**(rendezvous daemon)，它负责根据消息的主题发送和传递消息。无论何时发布了一条消息，都会把它多播给网络上每个运行Rendezvous守护进程的主机。一般来说，多播使用底层网络提供的功能实现，例如IP多播或硬件广播。

订阅某个主题的进程把它们的订阅传递给它们本地的守护进程。守护进程构造一个(进程，主题)表以及一些条目。每当一个关于主题S的消息到达时，守护进程简单地查看自己表中的本地订阅者，并把这个消息转发给每个订阅者。如果没有S的订阅者，将立即丢弃这个消息。

当像TIB/Rendezvous中那样使用多播时，没有理由不对订阅进行精心设计，并且应该使其超越目前使用的字符串比较。这里的关键问题是：由于消息无论如何都会被转发给每个结点，完全可以在本地完成发布数据与订阅之间可能很复杂的匹配，而无需进一步的网络通信。不过，如我们后面将讨论的，无论何时需要跨广域网进行匹配，都需要一些简单的比较规则。

13.2.3 对等体系结构

大多数基于协作的系统遵循的传统体系结构都会遭受一些可伸缩性问题(尽管它们的商业供应商往往另有说辞)。显然,具有用于匹配订阅与已发布数据的中央服务器并不能扩展到超过几百个客户。此外,使用多播要求采用特殊的措施以超越局域网的界限。而且,如果要保证可伸缩性,可能需要对描述订阅和数据项采取进一步的限制。

人们做了大量的研究工作,以使用对等技术实现基于协作的系统。目前存在一些针对这些情况的直观的技术,其中使用了一些关键字,因为它们可以被散列成用于已发布数据的唯一标识符。这种方法也被用于把(属性,值)对映射到标识符。在这些情况下,匹配就简化成直接查找一个标识符,在基于 DHT 的系统中可以高效地实现这种操作。如(Tam 和 Jacobsen 2003)所说明的,这种方法对于更传统的发布/订阅系统工作得很好,对于生成通信系统也是如此(Busi 等 2004)。

对于更精心设计的匹配模式,事情就变得复杂了。众所周知的困难是：需要支持一些取值范围,并且只有极少的提议存在。下面,我们将讨论这样一个提议,它是由作者之一及其同事设计的(Voulgaris 等 2006)。

13.2.3.1 示例：基于闲聊的发布/订阅系统

考虑一个发布/订阅系统,其中可以通过 N 个属性 $a_1, \cdots, a_N$ 描述数据项,这些属性的值可以直接映射到一个浮点数。这些值包括(例如)浮点数、整数、枚举、布尔值和字符串。订阅 s 采用(属性,值/范围)对的元组形式,比如：

$$s = \langle a_1 \rightarrow 2.0, a_4 \rightarrow [0.0, 0.5) \rangle$$

在这个示例中,s 指定 a_1 应该等于 3.0,a_4 应该位于区间[0.0,0.5)内。其他属性允许取任意值。为了清楚起见,假定每个结点 i 只输入一个订阅 s_i。

注意,每个订阅 s_i 实际上指定了浮点数的 N 维空间中的一个子集 S_i。这样一个子集也被称为超空间。对于整个系统,感兴趣的只是其描述属于这些超空间的并集 $S = \bigcup S_i$ 内的已发布数据。其整体思想是：自动把 S 分成 M 个分离的超空间 $S_1, \cdots, S_M$,使得每个订阅完全属于订阅超空间 S_i 之一,这些超空间一起涵盖了所有的订阅。更正式地讲,我们具有：

$$(S_m \cap S_i \neq \phi) \Rightarrow (S_m \subseteq S_i)$$

而且,系统使 M 保持最小,从这个意义上讲,没有分区具有更少的部分 S_m。要表达的整体思想是：对于每个超空间 S_m,恰好有这些结点 i,其中 $S_m \subseteq S_i$。在这种情况下,当发布数据项时,系统只需要查找数据项所属于的 S_m,从此它就可以把数据项转发给关联的结点。

此时,结点将使用一种流行的协议定期交换订阅。如果两个结点 i 和 j 注意到它们各自的订阅相交,即 $S_{ij} \equiv S_i \cap S_j \neq \phi$,它们将记录这一事实,并且保存指向对方的引用。如果它们发现第三个结点 k 具有 $S_{ijk} \equiv S_{ij} \cap S_k \neq \phi$,这三个结点将相互连接,使得可以高效地传播来自 S_{ijk} 中的数据项 d。注意：如果 $S_{ij} - S_{ijk} \neq \phi$,那么结点 i 和 j 将维持它们的相互引用,但是现在将用 $S_{ij} - S_{ijk}$ 严格地关联它。

实质上讲,我们正在寻求的是把结点群集到 M 个不同组中的方式,使得当且仅当结点 i 和 j 的订阅 S_i 和 S_j 相交时,才把它们归于同一个组中。而且,应该把同一个组中的结点组

织将一个覆盖网络中，它将允许在与这个组关联的超空间中高效地传播数据项。图 13.5 中描述了这种针对单个属性的情形。

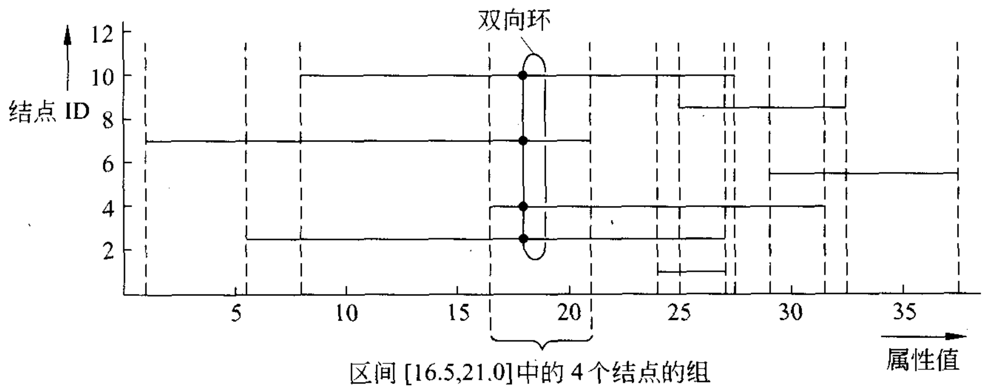


图 13.5　在对等发布/订阅系统中组织结点以支持范围查询

这里，我们看到了总共 7 个结点，其中用于结点 i 的水平线指示了它对单个属性的值的兴趣范围。另外还可看到把结点组织进属性值的不相连的兴趣范围中。例如，结点 3、4、7 和 10 将被组织在一起，表示区间[16.5，21.0]。对于任何具有这个范围中的一个值的数据项，都只应该把它传播给这 4 个结点。

为了构造这些组，把结点组织进基于闲聊的非结构化网络中。每个结点维护一个指向其他邻居的引用列表（即**局部视图**(partial view)），如第 2 章所述，该结点定期与其邻居之一交换信息。这种信息交换允许结点随机获悉系统中的其他结点。每个结点都了解它利用重叠兴趣（即利用交叉订阅）发现的其他结点。

在某个时刻，每个结点 i 一般将拥有指向其他具有重叠兴趣的结点的引用。作为与结点 j 交换信息的一部分，结点 i 按标识符对这些结点进行排序，并选择一个具有最小标识符 $i_1>j$ 的结点，使得其订阅与结点 j 的订阅重叠，即 $S_{j,i_1} \equiv S_{i_1} \cap S_j \neq \phi$。

下一个要选择的结点是 $i_2 > i_1$，使得其订阅也与 j 的订阅重叠，但是仅当它包含尚未被结点 i_1 涵盖的元素时才会如此。换句话说，我们应该具有 $S_{j,i_1,i_2} \equiv (S_{i_2} - S_{j,i_1}) \cap S_j \neq \phi$。这个过程将重复进行，直到与结点 i 之间具有重叠兴趣的所有结点都被检查到为止，从而得到一个有序列表 $i_1 < i_2 < \cdots < i_n$。注意：结点 i_k 在这个列表中，因为它涵盖了结点 i 和 j 的公共兴趣的区域 R，而它尚未被标识符比 i_k 小的结点共同涵盖。实际上，结点 i_k 是结点 j 应该把数据项转发到的属于这个唯一区域 R 中的第一个结点。可以把这个过程扩展到让结点 i 构造一个双向环。图 13.5 中也显示了这样一个环。

无论何时发布数据项 d，都会尽可能快地把它传播给对它感兴趣的任何结点。事实证明，利用每个结点提供的信息，很容易找到对 d 感兴趣的结点 i。从那时起，结点 i 需要简单地沿着 d 所属于的特定范围的订阅者的环转发 d。为了加快传播 d 的速度，还要为每个环维护一些捷径。在(Voulgaris 等 2006)中可以找到详细信息。

13.2.3.2　讨论

有一种方法有些类似于这种基于闲聊的解决方案，这是由于它尝试找到属性值所涵盖空间的一个分区，但它使用了(Gupta 等 2004)中描述的基于 DHT 的系统。在(Bharambe

2004）中描述的另一个提议中，通过单独的进程 P_i 处理每个属性 a_i，a_i 反过来又跨多个进程划分其属性的范围。当发布数据项 d 时，将它转发给每个 P_i，随后将其存储在负责 d 的 a_i 值的进程中。

所有这些方法都可用于说明把有价值的发布/订阅系统映射到对等网络时的复杂性。实质上讲，这种复杂性源于以下事实：天生就很难以一种分散式的方式建立一种支持搜索的基于属性的命名系统。在讨论复制时，我们将再次遇到这些困难。

13.2.4 移动性和协作

在文献中受到人们相当多的注意的一个主题是：如何把发布/订阅解决方案与结点移动性结合起来。在许多情况下，假定有一个固定的基础结构，它带有一些用于移动结点的接入点。在这些假设之下，问题变成了如何确保不会把已发布的消息给那些切换接入点的订阅者提供多次。这个问题的一种实用的解决方案是：让订阅者记录它们已经接收到的消息，并简单地丢弃重复的消息。一种更复杂的替代解决方案包含路由器，用于记录哪些消息已经发送给哪些订阅者（Caporuscio 等 2003）。

示例：Lime

就生成通信而言，已经提出了几种解决方案来操作一个共享的数据空间，其中的（一些）结点是可移动的。这种情况下的一个规范的示例是 Lime（Murphy 等 2001），它与我们以前讨论过的 JavaSpace 模型非常相似。

在 Lime 中，每个进程都有它自己关联的数据空间，但是当进程彼此邻近以便可以把它们联系在一起时，它们的数据空间就变成共享的了。从理论上讲，使进程联系在一起可能意味着在共同的底层网络中有一个路由器，它允许两个进程交换数据。不过，在实际中，它要么意味着两个进程暂时位于相同的物理主机上，要么意味着它们各自的主机可以通过一条（单个跳段）无线链路相互通信。正式地讲，这些进程应该是同一个组中的成员，并且使用相同的组通信协议。

相联系的进程的本地数据空间构成了一个瞬时共享的数据空间，它将允许进程之间交换元组，如图 13.6 所示。例如，当进程 P 执行一个 write 操作时，就把关联的元组存储在进程的本地数据空间中。从原理上讲，它将保留在那里，直到出现一个匹配的 take 操作，它可能是位于和 P 相同的组中的另一个进程执行的。这样，我们实际上是在与一个完全分布式的共享数据空间打交道的事实对于参与的进程就是透明的。不过，Lime 还允许通过准确指定打算把元组发送给哪个进程来打破这种透明性。此外，read 和 take 操作还可以具有额外的参数，用于指定期望从哪个进程接收元素。

为了更好地控制如何分布元组，数据空间可以执行所谓的**反应**（reaction）。当在本地数据空间中找到与给定模板匹配的元组时，反应指定了一种要执行的动作。每当数据空间发生变化时，就会随机选择一个可执行的反应，这导致会导致对数据空间的进一步的修改。反应跨越当前共享的数据空间，但是有几个限制，用以确保它们会被高效执行。例如，就弱反应来说，如果匹配的数据仍然可访问，则只能保证关联的动作最终会被执行。

在 TOTA 中进一步应用了反应的思想，其中每个元组都有一个关联的代码段，准确指

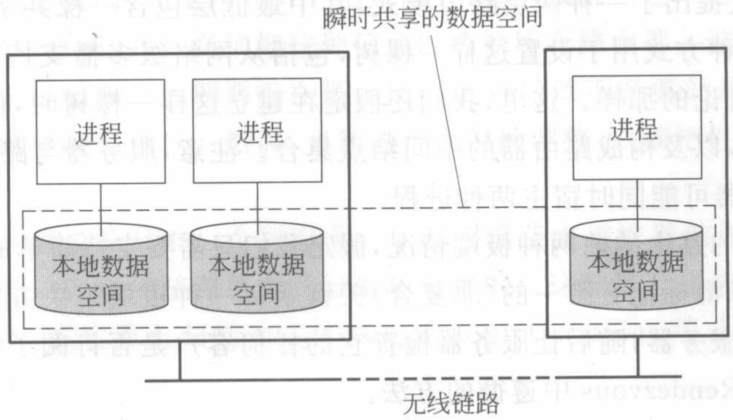


图 13.6 在 Lime 中瞬时共享本地数据空间

出应该如何在数据空间之间移动元组，可能还包括转换(Mamei 和 Zambonelli 2004)。

13.3 进 程

关于发布/订阅系统中使用的进程确实没有任何特殊之处。在大多数情况下，需要部署高效的机制，以便在可能很大的数据集合中执行搜索。其主要问题是：设计出可以在分布式环境中很好地工作的模式。我们将在下面讨论一致性和复制时回到这个问题上来。

13.4 通 信

许多发布/订阅系统中的通信相对比较简单。例如，在几乎所有基于 Java 的系统中，所有的通信都是通过远程方法调用进行的。当发布/订阅系统分散在整个广域系统中时，需要处理的一个重要问题是：已发布的数据只应该到达相关的订阅者那里。一种解决方案是：如前所述，通过使用自组织的方法，可以自动群集对等系统中的结点，然后在每个群集中传播数据项。另一种解决方案是部署基于内容的路由。

13.4.1 基于内容的路由

在**基于内容的路由**(content-based routing)中，假定系统构建在点对点网络之上，其中的消息是在结点之间显式路由的。这种设置的关键特性是：路由器可以通过考虑消息的内容来决定路由。更确切地讲，假定每条消息都携带有其内容的描述，并且这种描述可以用于隔离路由器，如果它所知道的路由器不会通向对消息感兴趣的接收方，就会隔离它们。

(Carzaniga 等 2004)提出了一种针对基于内容的路由的实用方法。考虑一个发布/订阅系统，其中包含 N 个服务器，客户(即应用程序)可以向它们发送消息，或者通过它们阅读进入的消息。我们假定为了阅读消息，应用程序将在之前给服务器提供它感兴趣的数据类型的描述。当相关数据到达时，服务器反过来又会通知客户。

Carzaniga 等人提出了一种两层路由模式，其中最低层包含一棵共享的广播树，它连接 N 个服务器。有多种方式用于设置这样一棵树，包括从网络级多播支持到应用程序级多播树，如第 4 章中所讨论的那样。这里，我们还假定在建立这样一棵树时，使之具有 N 个作为终端结点的服务器，以及构成路由器的中间结点集合。注意，服务器与路由器之间只有逻辑上的区别：单个机器可能同时宿主两种进程。

考虑基于内容的路由的前两种极端情况，假定我们只需要支持简单的基于主题的发布/订阅，其中每条消息都标记有唯一的（非复合）关键字。一种极端解决方案是把每条已发布的消息发送给每个服务器，随后让服务器检查它的任何客户是否订阅了该消息的主题。实质上讲，这是 TIB/Rendezvous 中遵循的方法。

另一种极端解决方案是让每个服务器把它的订阅广播给所有其他的服务器。因此，每个服务器将能够编译一份（主题，目的地）对的列表。然后，无论何时应用程序提交一条关于主题 s 的消息，它的关联的服务器都将为该消息预先计划目的服务器。当消息到达路由器时，后者可以使用该列表来决定消息应该遵循的路径，如图 13.7 所示。

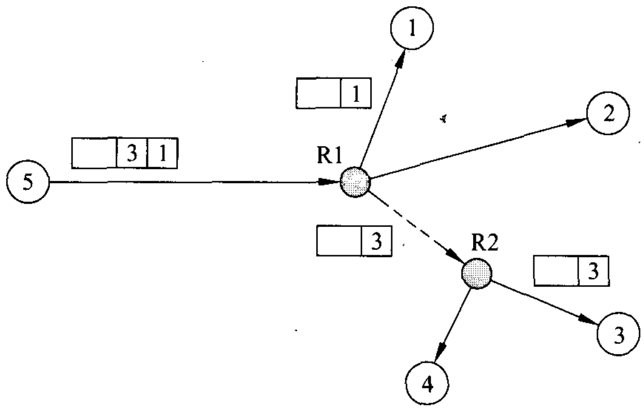


图 13.7　从未用过的基于内容的路由

采用后一种方法作为我们的起点，我们可以细化路由器的能力，以便决定把消息转发到哪里。此时，每个服务器都在整个网络中广播它的订阅，以便路由器可以创建**路由过滤器**(routing filter)。例如，假定图 13.7 中的结点 3 订阅了属性 a 位于区间[0,3]中的消息，但是结点 4 想要 a∈[2,5]的消息。在这种情况下，路由器 R_2 将把路由过滤器创建为一张表，对于它的每条外出链路（在这里有三条链路：一条通往结点 3，一条通往结点 4，还有一条通往路由器 R_1），该表中都会有一个对应的条目，如图 13.8 所示。

接口	过滤器
通往结点 3	a ∈ [0,3]
通往结点 4	a ∈ [2,5]
通往路由器 R_1	（未指定）

图 13.8　部分填充的路由表

更有趣的是在路由器 R_1 上发生的事情。在这个示例中，来自结点 3 和 4 的订阅规定：a 位于区间[0,3]∪[2,5]=[0,5]内的任何消息都应该沿着路径转发给路由器 R_2，并且 R_1 恰好要把这种信息存储在其路由表中。不难想象这可以支持更复杂的订阅复合。

这个简单的示例还说明，无论何时一个结点离开系统，或者当它不再对特定的消息感兴趣时，它都应该取消其订阅，并且必须把这种信息广播给所有路由器。这种取消反过来又可能导致调整多个路由过滤器。后期调整在最坏情况下将导致不必要的通信量，因为可能沿着不再有订阅者的路径转发消息。然而，必须进行及时的调整，以便把性能保持在可接受的级别上。

关于基于内容的路由的其中一个问题是：尽管创建路由过滤器的原理很简单，但是确定必须沿着其转发进入消息的链路可能是计算机密集的。这种计算复杂性来自于将属性值与订阅相匹配的实现，它实质上归结为逐条目比较。在(Carzaniga 等 2003)中描述了如何高效地执行这种比较。

13.4.2 支持复合订阅

迄今为止介绍的示例都是对路由表的简单扩展。当订阅采用(属性，值/范围)对的矢量形式时，这些扩展就足够了。不过，通常需要更复杂地表示订阅。例如，表示订阅的**复合**(composition)可能比较方便，其中一个进程在单个订阅中指定它感兴趣的是完全不同类型的数据项。举例说明，一个进程可能希望查看关于来自 IBM 的库存的数据项以及关于它们的收入的数据，但是只发送一种类型的数据项是无用的。

为了处理订阅复合，(Li 和 Jacobsen 2005)提出设计类似于规则数据库的路由器。实际上，订阅被转换成一些规则，它们指出了在哪些条件下应该转发已发布的数据，并且应该沿着哪些外出链路转发它们。不难想象这可能导致基于内容的路由模式，它们远比上述的路由过滤器更高级。支持订阅复合与基于协作的系统中的命名问题密切相关，我们接下来将讨论命名。

13.5 命　　名

让我们现在把注意力更多地放在基于协作的系统中的命名上。迄今为止，我们通常假定每个已发布的数据项都具有关联的 n 个(属性，值)对的矢量，并且进程可以通过在这些属性值之上指定谓词来订阅数据项。一般来讲，这种命名模式很容易使用，尽管系统在可以使用的属性类型、值和谓词方面有所差别。

例如，对于 JavaSpace，我们看到实质上只支持相等性比较，尽管可以相对容易地以特定于应用程序的方式对此进行扩展。此外，许多商业发布/订阅系统只支持相当原始的字符串比较运算符。

我们已经提过的问题之一是：在许多情况下，我们不能简单地假定每个数据项都标记有所有属性的值。特别是，我们将看到数据项只有一个关联的(属性，值)对，在这种情况下，也可以把它称为**事件**(event)。支持订阅事件(特别是复合事件)在很大程度上规定了在发布/订阅系统中关于命名问题的讨论。应该把我们迄今为止所讨论的方法视作支持分布式系统中的协作的更原始的方式。我们现在将更深入地探讨事件和事件复合。

当处理复合事件时，我们需要考虑两个不同的问题。第一个是描述复合。这种描述构

成了订阅的基础。第二个问题是如何收集（原始）事件，然后把它们与订阅相匹配。(Pietzuch 等 2003)提出了一种用于分布式系统中的事件复合的一般框架。我们把这个框架用作讨论的基础。

13.5.1 描述复合事件

让我们首先考虑复合事件的一些示例，以便对我们可能需要处理的复杂性有一个更好的了解。图 13.9 显示了日益复杂的复合事件的一些示例。在这个示例中，R4.20 可能是装有空调的、安全的计算机室。

示例	描述
S_1	当 R4.20 房间空闲时请发送通知
S_2	当 R4.20 房间空闲并且房门未锁时请发送通知
S_3	当 R4.20 房间空闲 10 秒钟同时房门未锁时请发送通知
S_4	当 R4.20 房间的温度每 30 分钟升高超过 1℃时请发送通知
S_5	当 R4.20 房间内的平均温度在过去 30 分钟内超过 20℃时请发送通知

图 13.9 分布式系统中的事件示例

前两种订阅相对比较容易。S_1 是一个可以通过原始离散事件处理的示例，而 S_2 则是两个离散事件的简单复合。订阅 S_3 更复杂，因此它要求系统还可以报告与时间相关的事件。如果订阅涉及计算斜率(S_4)或平均值(S_5)所需要的聚合值，事情将会变得更复杂。注意：就 S_5 来说，我们需要持续不断地监视系统，以按时发送通知。

用于分布式系统中事件复合语言背后的基本思想是：支持根据原始事件表述订阅。在它们的框架中，Pietzuch 等人为有限状态机(finite-state machine，FSM)的扩展类型提供了一种相对比较简单的语言。这些扩展允许规定状态中的暂时停留的时间，以及生成新的(复合)事件。它们的语言的精确细节对我们这里的讨论并不重要。重要的是可以把订阅转换成 FSM。

为了给出一个示例，图 13.10 显示了用于图 13.9 中的订阅 S_3 的 FSM。通过时间状态给出特例，用标签“t＝10s”指示，它指定如果没有在 10 秒钟内锁上房门就完成向最终状态的转换。

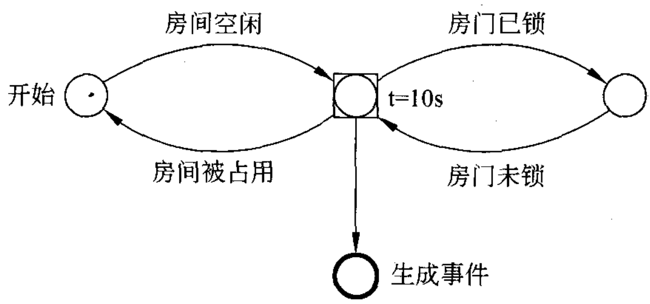


图 13.10 用于图 13.9 中的订阅 S_3 的有限状态机

还可以描述更复杂的订阅。一个重要方面是：通常可以把这些 FSM 分解成一些较小

的 FSM,这些 FSM 相互之间通过传递事件进行通信。注意:这种事件通信通常会触发执行事件的 FSM 上的一个状态转换。例如,假定当我们确定 R4.20 房间内不再有人(并且已锁上房门)时,我们想在 2 秒钟后自动关闭该房间内的灯。在这种情况下,如果我们让图 13.10 中的 FSM 为将触发照明设备的第二个 FSM 生成一个事件,就可以重用前一个 FSM,如图 13.11 所示。

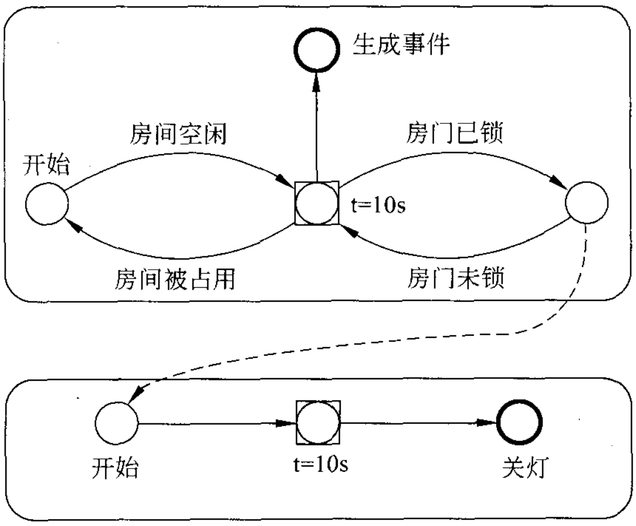


图 13.11 两个耦合的 FSM

这里的重要事项是:可以在分布式系统中将这两个 FSM 实现为单独的进程。在这种情况下,用于控制照明设备的 FSM 将订阅一个复合事件,当 R4.20 房间空闲并且房门已锁上时将触发该事件。这将导致我们接下来将讨论的分布式检测器。

13.5.2 匹配事件与订阅

现在考虑一种支持复合事件的发布/订阅系统。每个订阅都是以一个可以被转换成有限状态机(FSM)的表达式的形式提供的。状态转换实质上是由发生的原始事件触发的,比如离开房间或锁上门。

为了匹配事件与订阅,我们可以遵循一个简单的、以前未用过的实现,其中每个订阅者都运行一个进程,它实现与其订阅关联的有限状态机。在这种情况下,与特定订阅相关的所有原始事件都必须转发给订阅者。显然,这一般不是非常高效。

一种好得多的解决方案是考虑订阅的完整集合,并且把订阅分解成相互通信的有限状态机,使得在不同订阅之间共享其中一些 FSM。这种共享的一个示例如图 13.11 所示。这种用于处理订阅的方法导致了所谓的**分布式事件检测器**(distributed event detector)。注意:事件检测器的分布本质上类似于多种命名系统中的名称的分布式解析。原始事件导致相对简单的有限状态机中的状态转换,这反过来又触发了复合事件的生成。后者然后又可以导致其他 FSM 中的状态转换,这又可能导致进一步的事件生成。当然,事件将转换成消息,然后通过网络把这些消息发送给订阅它们的进程。

除了通过共享进行优化之外,把订阅分解成相互通信的 FSM 还具有优化网络使用的潜在优点。再次考虑与上述监视计算机室相关的事件。假定只有对复合事件感兴趣的进

程。把这些事件组合成与计算机室联系紧密是有意义的。这种布置将防止不得不跨网络发送原始事件。而且，当考虑图13.9时，我们看到当通知房间空闲10秒钟同时房门未锁时，我们可能只需要发送警报。例如，与(未)锁房门相比，这种事件一般很少发生。

把订阅分解到分布式事件检测器中，然后在整个分布式系统中最佳地放置它们，这个主题仍然需要进行大量的研究。例如，关于订阅语言的具体表现仍无定论，特别是实现的表达力和效率之间的折衷仍将吸引许多注意力。在大多数情况下，语言的表达力越强，就越不可能出现一种高效的分布式实现。目前的提议(比如Demers等人(2006)以及Liu和Jacobsen(2004)的提议)就证实了这一点。可能要花费数年的时间才能把这些技术应用于商业发布/订阅系统。

13.6 同　步

基于协作的系统中的同步一般受限于支持生成通信的系统。当只使用一个服务器时，事件相对比较直观。在这种情况下，在元组可用之前可以简单地阻塞进程，但是删除元组可以使事情变得更简单。当复制共享的数据空间并在多个服务器之间分布它们时，事件将变得复杂，接下来我们将讨论这一点。

13.7 一致性和复制

在基于协作的系统(特别是那些用于生成通信的系统)的可伸缩性中，复制扮演着一种关键角色。下面，我们首先考虑在许多系统(如JavaSpace)中探索过的一些标准方法。接下来，我们将描述一些最近的成果，它们允许依据元组的访问模式动态、自动地布置元组。

13.7.1 静态方法

支持生成通信的系统的分布式实现往往需要特别注意。我们重点关注JavaSpace服务器的可能的分布式实现，通过这种实现，可能跨多个机器分布和复制元组实例的集合。Rowstron(2001)给出了有关基于元组的运行时系统的实现技术的概述。

一般考虑

一个高效的JavaSpace分布式实现必须解决以下两个问题：

(1) 如何在不进行大量搜索的情况下模拟相关寻址。

(2) 如何把元组分布到多个机器上，并在以后定位它们。

要解决这两个问题，关键是要认识到每个元组是一个类型化的数据结构。把元组空间分为多个子空间，每个子空间中的元组具有相同的类型，这样做可以简化程序设计，并使优化措施成为可能。例如，由于元组是类型化的，所以可以在编译时决定write、read或take操作的对象。这种划分意味着只需要搜索元组实例集合的一部分。

另外，每个子空间可以把其第 i 个元组字段（或该字段的一部分）作为散列键来构造一个散列表。回忆可知，元组实例中的每个字段是对一个对象的编组引用。JavaSpace 没有预先定义应该如何进行编组。因此，实现可以决定在对引用进行编组时，使前几个字节用作待编组对象类型的标识符。在执行对 write、read 或 take 操作的调用时，可以计算第 i 个字段的散列函数，以找到元组实例在表中的位置。如果知道子空间和表的位置，就可以免去所有的搜索过程。当然，在 read 或 take 操作的第 i 个字段为 NULL 时，不可能进行散列处理，因此一般需要对子空间进行完整的搜索。不过，通过谨慎地选择进行散列处理的字段，通常是可以避免搜索的。

还使用了其他的优化措施。例如，上面描述的散列模式把一个给定子空间的元组分布到一些箱子（bin）中，从而把搜索限制在单个箱子中。可以把不同的箱子放置在不同的机器上，这样既可以更广泛地分散负载，也可以更好地利用局部化的优势。如果散列函数是对类型标识符取机器数目的模，那么箱子的数目是随系统规模线性伸缩的（Bjomson 1993）。

在计算机网络上，最佳的选择取决于通信的体系结构。如果存在可靠的多播，那么可以选择把所有子空间完全复制到所有的机器上，如图 13.12 所示。当进行一个 write 操作时，就广播新的元组实例，并把它写入每个机器上的相应子空间中。为了进行一个 read 或 take 操作，对本地子空间进行搜索。不过，由于成功完成的 take 操作会把元组实例从 JavaSpace 中删除，因此需要使用一个删除协议来把该元组实例从所有机器上删除。为了防止竞争和死锁，可以使用一个两阶段提交协议。

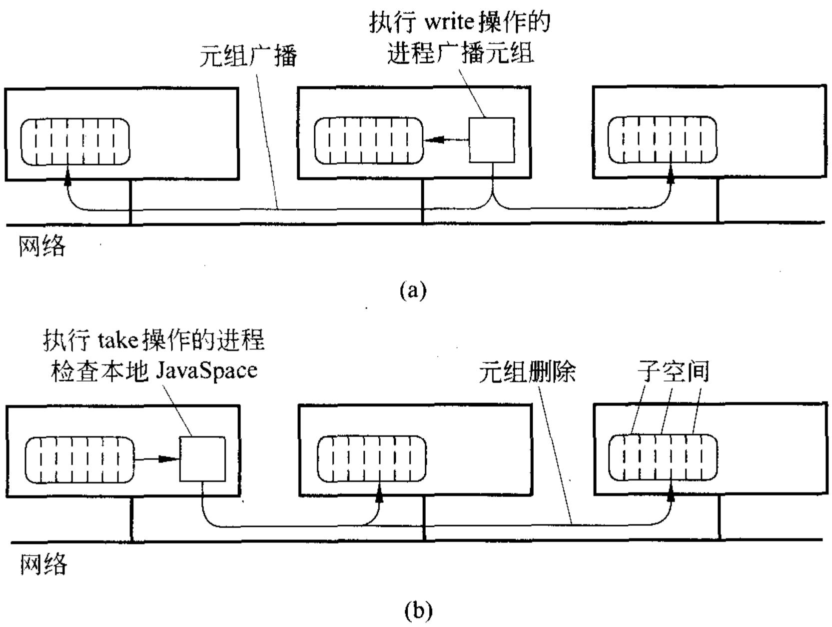
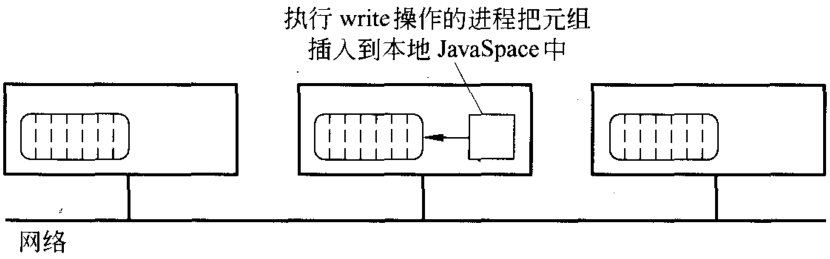


图 13.12　可以把一个 JavaSpace 复制到所有机器上。虚线表明把 JavaSpace 划分为一些子空间
（a）在执行 write 操作时广播元组；（b）read 是本地操作，但调用 take 删除一个元组实例时必须进行广播

这种设计是直观的，但是当系统在元组实例数目和网络规模方面增长时，它的可伸缩性并不是很好。例如，在广域网中实现该方案的代价是极为昂贵的。

相反的设计方案是在本地进行 write 操作，把元组实例仅仅存储在生成它的机器上，如图 13.13 所示。在进行 read 或 take 操作时，进程必须广播模板元组。每个接收方然后都会

检查自己是否有与之匹配的元组实例，如果有，将发回一个应答消息。



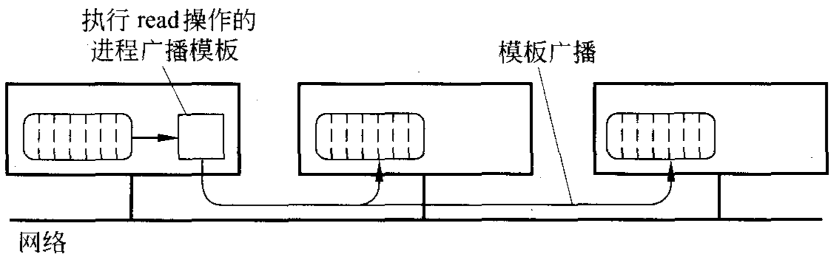


图 13.13 未复制的 JavaSpace

(a) 在本地执行 write 操作；(b) read 或 take 操作需要广播模板元组，以找到元组实例

如果不存在元组实例，或者保存元组的机器没有收到广播，请求的机器将无限地重新广播请求，同时逐渐增加两次广播之间的时间间隔，直到出现合适的元组实例并且请求得到满足为止。如果发送了两个或多个元组实例，就会像本地 write 操作那样进行处理，并且实际上会把它们从各自所在的机器上移到发出请求的机器上。事实上，运行时系统甚至可以自主地移动元组，从而进行负载平衡。(Carriero 和 Gelemter 1986)在局域网上实现 Linda 元组空间时使用了这种方法。

可以结合使用这两种方法，得到部分复制的系统。举一个简单的例子，设想所有的机器在逻辑上构成了一个矩形网格，如图 13.14 所示。当机器 A 上的一个进程希望进行一个 write 操作时，它把元组广播（或用点到点消息发送）给它所在网格行中的所有机器。当机器 B 上的一个进程希望读或取出一个元组实例时，它把模板元组广播给它所在列中的所有机器。根据几何学原理，总会正好有一台机器同时看到元组实例和模板元组（在本例中是机器 C），该机器进行匹配，并把该元组实例发送给请求它的进程。这种方法类似于我们在第 7 章中讨论的基于法定数量(quorum-based)的复制。

迄今为止讨论的实现具有严重的可伸缩性问题，因为在对元组空间插入或删除元组时都会用到多播。元组空间的广域实现尚不存在。在最好情况下，多个不同的元组空间可以共存于单个系统中，其中每个元组空间本身在单个服务器或一个局域网中实现。这种方法用于 PageSpaces(Ciancarini 等 1998)和 WCL(Rowstron 和 Wray 1998)中。在 WCL 中，每个元组空间服务器负责整个元组空间。换言之，进程总被定向到恰好一个服务器。不过，可以把一个元组空间迁移到一个不同的服务器上以改善性能。如何开发元组空间的高效广域实现仍然是一个未知的问题。

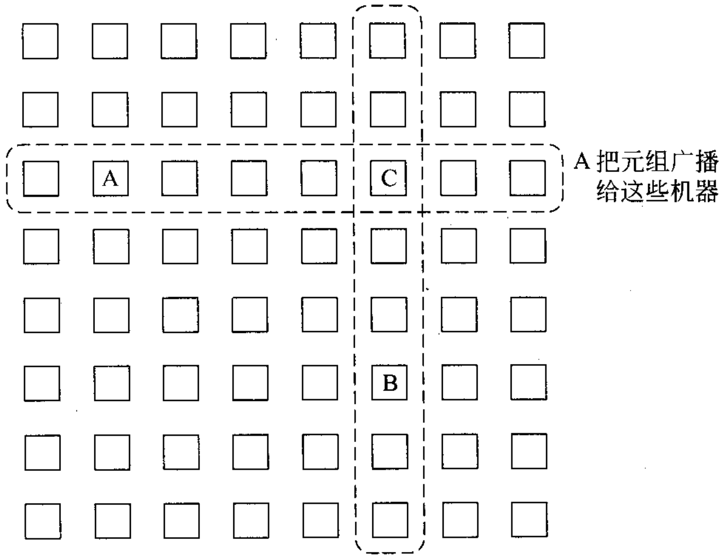


图 13.14 部分广播元组和模板元组

13.7.2 动态复制

基于协作的系统中的复制一般受限于用于并行应用程序(就像上述那些应用程序)的静态策略。在商业应用程序中,我们还会看到一些相对比较简单的模式,其中整个数据空间或者数据集的静态预定义的部分受单个策略支配(GigaSpaces 2005)。受 Globule 中 Web 文档的细粒度复制的鼓舞,当在数据空间中存储的不同类型的数据之间有区别地进行复制时,也可以实现性能改进。这种区分受 GSpace 的支持,我们将在本节中简要讨论它。

13.7.2.1 GSpace 概述

GSpace 是一种构建于 JavaSpace 之上的基于协作的分布式系统(Russello 等 2004、2006)。出于两个不同的原因而对 GSpace 中的元组进行分布和复制,这两个原因是:改进性能和可用性。这种方法中的一个关键元素是把要考虑的问题隔离开:为提高可用性而进行复制的元组与为改进性能而进行复制的元组之间可能需要遵循不同的策略。因此,GSpace 的体系结构被建立成支持多种复制策略,因而不同的元组可能遵循不同的策略,如图 13.15 所示。

它的基本工作原理相对比较简单。每个应用程序都会被提供一个接口,它带有 read、write 和 take 接口,类似于 JavaSpace 所提供的接口。不过,每个调用都会被本地调用处理程序获取,该处理程序查找特定的调用应该遵循的策略,基于作为调用的一部分传递的元组/模板的类型和内容选择策略。每个策略都由模板标识,类似于如前所述的在其他基于 Java 的共享数据空间中使用模板选择元组的方式。

这种选择的结果是一个指向分布管理器的引用,它实现了相同的接口,但是现在要根据特定的复制策略实现它。例如,如果实现了主/从策略,就可能通过立即从本地可用的数据

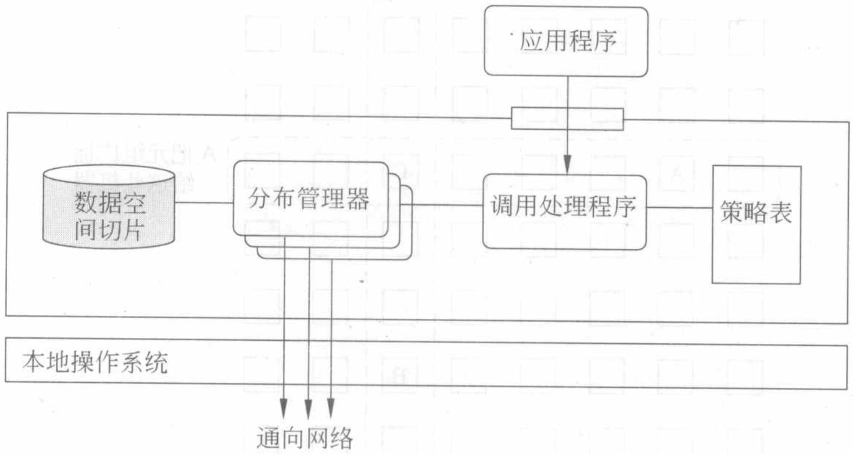


图 13.15 GSpace 核心的内部组织结构

空间中读取一个元组来实现 read 操作。此外，write 操作可能要求分布管理器把更新转发给主结点，并等待一个确认，然后在本地执行操作。

最后，每个 GSpace 核心都有一个本地数据空间，称为切片，它被实现为 JavaSpace 的成熟的、非分布式的版本。

在这种体系结构中(为了清楚起见，没有显示其中一些组件)，可以在运行时添加策略描述符，此外，也可以更改分布管理器。这种设置允许对元组的分布和复制进行细粒度的调整，如(Russello 等 2004)中所说明的那样，与应用于数据空间中的所有元组的任何固定的全局策略相比，这种细粒度调整允许实现好得多的性能。

13.7.2.2 自适应性复制

不过，像 GSpace 这样的系统的最重要的方面是自动进行复制管理。换句话说，它不是让应用程序开发人员查明哪种策略组合是最佳的，更好的做法是让系统监视访问模式和行为，然后根据需要采用策略。

此时，GSpace 遵循与 Globule 中相同的方法：它持续不断地度量消耗的网络带宽、延迟和内存使用，并且依赖于其中哪些指标被认为是最重要的，把元组放置在不同的结点，并选择最合适的方式使副本保持一致。评估哪种策略对于给定的元组是最佳的这项工作通过中央协调器来完成，中央协调器简单地从构成 GSpace 系统的结点上收集迹线。

一个有趣的方面是：我们时不时可能需要从一种复制策略切换到另一种复制策略。可以用多种方式来执行这种转换。由于 GSpace 的目标是尽可能最佳地把机制与策略隔离开，因此它也可以处理不同的**转换策略**(transition policy)。默认情况是临时冻结对特定类型的元组的所有操作，删除所有的副本，并把元组重新插入到共享的数据空间中，但是现在将遵循最近选择的复制策略。不过，依赖于新的复制策略，也许可以使用执行转换的不同(并且更廉价的)方式。例如，当从无复制切换到主/从复制时，当首先访问从副本时，一种方法可能是懒惰地把元组复制到这些从副本上。

13.8 容 错 性

当考虑到容错性是任何分布式系统的基础时,令人稍感意外的是:在基于协作的系统中在容错性上面只投入了相对很少的注意力,这些系统包括基本的发布/订阅系统,以及那些支持生成通信的系统。在大多数情况下,注意力集中在确保高效数据传送的可靠性上,它实质上可以归结为保证可靠的通信。当期望中间件也存储数据项时,就像生成通信中那样,人们在可靠存储方面做了一些工作。让我们更详细地探讨这两种情况。

13.8.1 可靠的发布-订阅通信

在基于协作的系统中,其中已发布的数据项只与活动的订阅者相匹配,可靠的通信就扮演着一种至关重要的角色。在这种情况下,通常通过实现实际的发布/订阅软件下面的可靠多播系统来实现容错性。一般要注意几个问题。第一,以与基于内容的路由无关的方式建立可靠的多播信道。第二,需要处理进程容错性。让我们看看在 TIB/Rendezvous 中如何解决这些问题。

示例:TIB/Rendezvous 中的容错性

TIB/Rendezvous 假设底层网络的通信功能本质上是不可靠的。为了弥补这一不可靠性,无论何时一个 Rendezvous 守护进程向其他守护进程发布一条消息时,它都会把这条消息至少保存 60 秒。当发布一条消息时,守护进程把一个与主题无关的序号附加到这条消息上。接收消息的守护进程通过查看序号可以检测是否有消息丢失(回忆一下,消息会传递给所有守护进程)。当丢失了一条消息时,TIB/Rendezvous 会请求发布消息的守护进程重传该消息。

这种形式的可靠通信仍然不能防止消息可能会丢失。例如,如果一个接收消息的守护进程请求重传一条早在 60 秒之前发布的消息,则发布消息的守护进程对恢复这个已丢失的消息通常是无能为力的。在正常的情况下,TIB/Rendezvous 会通知负责发布和订阅的应用程序发生了一个通信错误,而错误可以留给应用程序以后处理。

TIB/Rendezvous 中通信的可靠性主要基于其底层网络提供的可靠性。最近,TIB/Rendezvous 也提供了可靠的多播功能,它使用(不可靠的)IP 多播作为底层的通信手段。TIB/Rendezvous 所遵循的模式是一种称为**实际通用多播**(Pragmatic General Multicast,PGM)的传输层多播协议,(Speakman 等 2001)中有对该协议的描述。我们将对 PGM 进行简要讨论。

PGM 并不绝对保证多播的消息最终一定能传送给每个接收方。在图 13.16(a)中,一条消息沿树进行多播,但是它并没有传送给两个接收方。PGM 依靠接收方来检测出它们丢失了消息,并向消息的发送方发送一个重传请求(即一个 NAK)。该请求沿着以消息发送方为根的树中的反向路径发送,如图 13.16(b)所示。当一个重传请求到达一个中间结点时,该结点可能已缓存了请求的消息,此时它将重传该消息。否则,该结点只是简单地把

NAK消息转发给通向发送方的下一个结点。发送方最终负责重传消息。

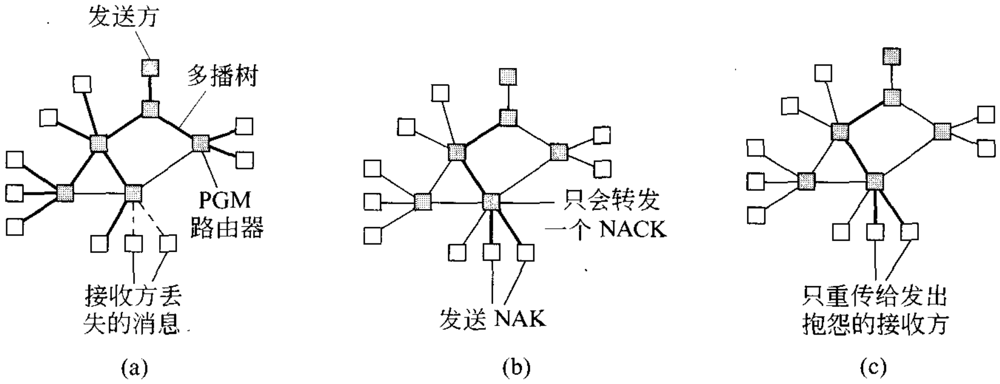


图13.16 PGM的原理

(a)沿多播树发送一条消息;(b)路由器只为每条消息传递一个NAK;(c)消息只重传给请求它的接收方

PGM采用许多措施来为可靠的多播提供可伸缩的解决方案。首先,如果一个中间结点接收到对同一条消息的多个重传请求,它将只把一个重传请求转发给发送方。通过这种方式来确保只有一个NAK会到达发送方,从而避免出现反馈爆炸的现象。在第8章讨论可靠多播的可伸缩性问题时我们已经遇到过这个问题。

PGM采取的另一个措施是记下NAK从接收方到发送方所经过的路径,如图13.16(c)所示。当发送方最后重传被请求的消息时,PGM使得消息只多播给那些请求重传的接收方。因此,已经成功接收到消息的接收方就不会受到干扰。

除了基本的可靠性模式和通过PGM实现的可靠多播外,TIB/Rendezvous还通过**经过验证的消息传送**(certified message delivery)提供更高的可靠性。在这种情况下,进程使用一条特殊的通信信道来发送或接收消息。该信道具有一个关联的**账簿**(ledger),用于记录发送和接收的经过证明的消息。希望接收这种消息的进程在消息发送方那里进行注册。实际上,这种注册允许信道处理Rendezvous守护进程不提供支持的更多可靠性问题。其中大多数问题对应用程序是隐藏的,并且由信道的实现进行处理。

当把账簿实现为一个文件时,那么即使在进程出现故障的情况下也可提供可靠的消息传送。例如,当一个接收进程崩溃时,在它恢复前丢失的全部消息都会存储在发送方的账簿中。一旦恢复,接收方只需与账簿联系,并请求重传丢失的消息。

为了能够屏蔽进程故障,TIB/Rendezvous提供了一种自动激活进程或停用进程的简单手段。这里,活动的进程通常会对所有进入的消息做出响应,而不活动的进程则不会这样。不活动的进程是只能处理特殊事件的运行中的进程,我们稍后将进行解释。

可以把进程组织为进程组,其中的每个进程都有一个与之关联的唯一等级。进程的等级由其加权(手工指定)决定,但是同一组中的两个进程不能具有相同的等级。对于每个组,TIB/Rendezvous试图令其中的一些进程处于活动状态,这些进程的数目是特定于组的,称为该组的**活动目标**(active goal)。在许多情况下,把活动目标设置为1,从而使得与每个组之间的所有通信都简化为第7章中讨论过的基于主版本的协议。

活动的进程一般会向组中的所有其他进程发送一条消息,以宣布自己仍然在运行。当不再发出这样的**心跳消息**(heartbeat message)时,TIB/Rendezvous中间件将自动激活当前不活动的最高等级的进程。激活通过回调一个action操作完成,每个组成员都应实现该操

作。同样，当一个先前崩溃的进程重新恢复并变成活动的时，当前最低等级的活动进程将自动变为不活动状态。

为了与活动进程保持一致，不活动进程在变为活动进程前需要采取特别的措施。一种简单的方法就是让不活动的进程订阅与任何其他组成员相同的消息。它会以通常的方式处理进入的消息，但是并不会发布任何反应。注意：这种模式类似于主动复制。

13.8.2　共享数据空间中的容错性

在处理生成通信时，事情将变得更复杂。同样如(Tolksdorf 和 Rowstron 2000)中所指出的，一旦需要在共享数据空间中纳入容错性，解决方法通常可能会变得效率如此低下，以至于只有集中式实现才是切实可行的。在这种情况下，将应用传统的解决方案，特别是结合使用中央服务器与检查点技术，中央服务器是使用简单的主-备份协议备份的。

一种替代方法是：通过跨多个机器放置数据项的副本，更积极地部署复制。在 GSpace 中采用了这种方法，它实质上部署了一些机制，它们与它用于通过复制改进性能的机制相同。此时，每个结点都会计算它的可用性，然后将其用于计算单个(复制的)数据项的可用性(Russello 等 2006)。

为了计算它的可用性，一个结点会定期把一个时间戳写到持久存储器中，允许它计算正常运行的时间以及停止工作的时间。更确切地讲，依据**平均失效时间**(mean time to failure，MTTF)和**平均修复时间**(mean time to repair，MTTR)来计算可用性：

$$\text{可用性节点}=\frac{MTTF}{MTTF+MTTR}$$

为了计算 MTTF 和 MTTR，一个结点简单地查看记录的时间戳，如图 13.17 所示。这将允许它计算故障之间的平均时间，从而得到如下可用性计算方式：

$$\text{可用性节点}=\frac{\sum_{k=1}^{n}(T_k^{start}-T_{k-1}^{end})}{\sum_{k=1}^{n}(T_k^{start}-T_{k-1}^{end})+\sum_{k=1}^{n}(T_k^{end}-T_k^{start})}$$

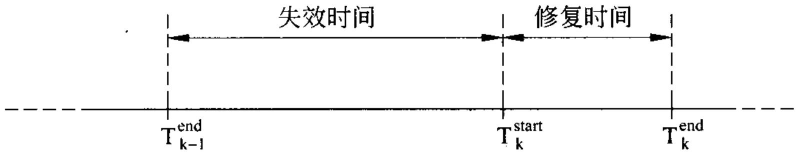


图 13.17　经过失效的结点的时间线

注意：必须定期记录时间戳，并且在发生崩溃时，只能把 T_k^{start} 视作最佳估计值。不过，这样计算得到的可用性将是悲观的，因为结点第 k 次崩溃的实际时间将稍晚于 T_k^{start}。此外，无需获取从开始算起的平均值，也可以只考虑最后的 N 次崩溃。

在 GSpace 中，每种数据项类型都有一个关联的主结点，它负责计算该类型的可用性。假定跨 m 个结点复制一个数据项，通过考虑 m 个结点中每个结点的可用性 a_i，可以计算它的可用性，从而得到：

$$可用性数据项 = 1 - \prod_{k=1}^{m}(1 - a_j)$$

只需考虑数据项的可用性以及所有结点的可用性，主结点就可以计算数据项的最佳布置，它将满足数据项的可用性需求。此外，它还可以考虑其他因素，比如带宽使用和CPU负载。注意：如果这些因素发生变化，数据项的布置随着时间的推移可能发生变化。

13.9 安 全 性

基于协作的系统中的安全性提出了一个困难的问题。一方面我们指出进程应该是引用性解耦合的，但是另一方面，我们还应该确保数据的完整性和保密性。这种安全性通常是通过安全的（多播）信道实现的，它实际上要求发送方和接收方能够相互进行身份验证。这种身份验证违反了引用性解耦合。

有不同的方法用于解决这个问题。一种常见的方法是建立一个中介网络，用于处理数据和订阅。客户进程然后将联系中介，中介然后负责身份验证和授权。注意：这种方法确实要求客户信任中介。不过，如我们稍后将看到的，通过区分不同的中介类型，客户不必信任组成系统的所有中介。

依据数据协作的性质，通常把授权转换成保密性问题。我们现在将遵循（Wang 等 2002）中提供的讨论，详细探讨这些问题。

13.9.1 保密性

许多分布式系统和基于协作的系统之间的一个重要区别是：为了提高效率，中间件需要检查已发布数据的内容。中间件不具有这种能力，它实质上只能把数据洪泛给所有潜在的订阅者。这提出了**信息保密性**（information confidentiality）的问题，它指的是如下事实：有时禁止中间件检查发布的数据是很重要的。可以通过端到端加密避开这个问题；路由实质上只能看到源和目的地的地址。

如果在构造发布的数据项时使每个数据项都包含多个字段，就有可能部署部分保密性。例如，可能需要在具有处于不同位置的分支机构的同一家事务所的代理之间传送关于房地产的数据，但是无需展示房地产的准确地址。为了允许基于内容的路由，可以对地址字段进行加密，但可以以明文形式发布房地产的说明。对此，（Khurana 和 Koleva 2006）提议使用逐字段加密模式，在（Bertino 和 Ferrari 2002）中介绍了它。在这种情况下，属于相同分支机构的代理将共享用于解密地址字段的密钥。当然，这违反了引用性解耦合，但是我们在后面将讨论这个问题的一种潜在的解决方案。

更难以处理的是：任何字段都不能以明文形式泄露给中间件。这时，余下的唯一解决方案是对加密的数据进行基于内容的路由。由于路由器只能看到加密的数据（可能基于逐个字段），将需要以可以进行部分匹配的方式对订阅进行编码。注意：部分匹配是路由器用于决定应该在哪条外出链路上转发已发布的数据项的基础。

这个问题非常接近于通过加密数据进行查询和搜索，显然，几乎不可能做到这一点。众

所周知，在维持高度的保密性的同时仍然提供合理的性能是非常困难的(Kantarcioglu 和 Clifton 2005)。其中一个问题是：如果使用逐字段加密，查明数据是什么就要容易得多。

不得不处理加密数据还引出了**订阅保密性**(subscription confidentiality)的问题，它指的是订阅可能不会泄露给中间件的事实。就基于主题的寻址模式来说，一种解决方案是简单地使用逐字段加密，并严格地基于各个字段应用匹配。部分匹配可以表示为其各个成分的集合。订阅者然后将发送这些成分的加密形式，并让路由器检查集合成员关系，Raiciu 和 Rosenblum(2005)也是这样建议的。事实证明，如果可以设计出一种高效的模式来表示区间，它甚至有可能支持范围查询。在(Li 等 2004a)中讨论了一种潜在的解决方案。

最后，**发布保密性**(publication confidentiality)也是一个问题。在这种情况下，我们接触的是更传统的访问控制机制，其中甚至不应该允许某些进程看到某些消息。在这些情况下，发布可能希望显式限制可能的订阅者组。在许多情况下，可以在发布和订阅应用程序的级别上不受限制地运用这种控制。不过，一种方便的做法是让中间件提供一种服务来处理这种访问控制。

解除发布者与订阅者之间的耦合

如果有必要对中间件保护数据和订阅，(Khurana 和 Koleva 2006)提出利用特殊的记账服务(accounting service，AS)，它实质上位于客户(发布者和订阅者)与实际的发布/订阅中间件之间。其基本思想是：解除发布者与订阅者之间的耦合，同时仍然提供信息保密性。在他们的模式中，订阅者注册它们对特定数据项的兴趣，然后像通常所做的那样路由它们。假定数据项包含加密的字段。为了允许进行解密，一旦应该把一条消息传送给订阅者，路由器就会把它传递给记账服务，并把它转换成只有订阅者才可以解密的消息。这种模式如图 13.18 所示。

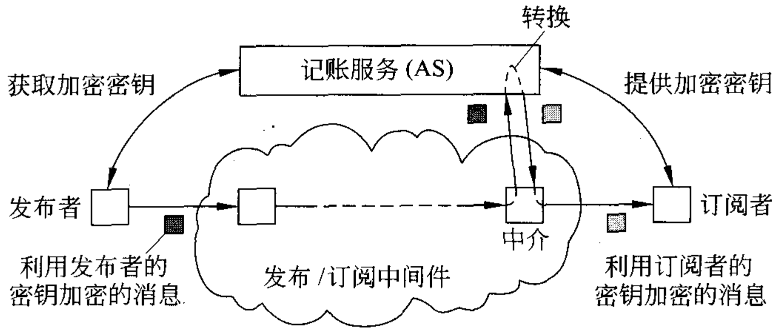


图 13.18 使用额外的可信服务解除发布者与订阅者之间的耦合

发布者在发布/订阅网络的任何结点(即中介)上注册它自身。中介把注册信息转发给记账服务，记账服务(AS)然后生成将被发布者使用的公钥并且分配它。当然，AS 自身会保留关联的私钥。当订阅者注册时，它会提供中介转发的加密密钥。有必要经过一个单独的身份验证阶段，以确保只有合法的订阅者可以注册。例如，一般不应该允许中介订阅已发布的数据。

在忽略许多细节的情况下，当发布数据项时，将由发布者对其关键字段进行加密。当数据项到达希望把它传递给订阅者的中介那里时，中介将请求 AS 转换消息，其方法是首先对它进行解密，然后利用订阅者提供的密钥对它进行加密。这样，中介永远不会知道应该保密

的内容，而与此同时，发布者与订阅者不必共享密钥信息。

当然，使记账服务本身可伸缩是至关重要的。可以采取多种措施，但是一种合理的方法是以与 Kerberos 类似的方式引入领域的概念。在这种情况下，可能需要使用外部记账服务的公钥重新加密传输中的消息来转换它们。有关详细信息，感兴趣的读者可以参考(Khurana 和 Koleva 2006)。

13.9.2 安全的共享数据空间

在保障共享数据空间的安全方面只做了极少的工作。一种常见的方法是简单地加密数据项的字段，并且仅当解密成功且内容与订阅匹配时才允许进行匹配。在(Vitek 等 2003)中描述了这种方法。这种方法的主要问题之一是：可能需要在发布者与订阅者之间共享密钥，或者应该使授权的订阅者知道发布者的解密密钥。

当然，如果共享数据空间是可信的(即允许实现数据空间的进程查看元组的内容)，事情就会变得简单得多。考虑到大多数实现只利用了单个服务器，实际中通常采用的方法是利用身份验证和授权机制扩展该服务器。

本章小结

基于协作的分布式系统在构建分布式应用程序方面发挥着重要的作用。其中大多数系统重点关注的是进程的引用性解耦合，即进程在通信时无需显式地彼此引用。此外，还可能提供现时性解耦合，其中的进程可以不必为进行通信而共存。

一组重要的基于协作的系统由那些像 TIB/Rendezvous 那样遵循发布/订阅范型的系统组成。在这种模型中，消息中不携带其接收方的地址，而是通过主题进行寻址。希望接收消息的进程应订阅某个特定的主题，中间件负责把消息从发布者传递给订阅者。

在更先进的系统中，订阅者可以通过已发布数据项的属性表述谓词。在这种情况下，我们正在与基于内容的发布/订阅系统打交道。出于效率起见，路由器可以安装过滤器，使得只通过那些已知有订阅者的外出链路转发已发布的数据就很重要。

另一组基于协作的分布式系统使用生成通信，它通过元组的共享空间实现。元组与记录类似，是一个类型化的数据结构。为了从一个元组空间中读取一个元组，进程通过提供一个模板元组指定要寻找的元组。匹配该模板的元组会被挑选出来，并返回给请求的进程。如果找不到匹配的元组，进程会阻塞。

基于协作的系统与其他分布式系统的不同在于：它们完全致力于为进程提供一种无须事先得知对方情况即可进行通信的便利方式。通信还能够以匿名的方式继续下去。这种方法的主要优点在于它的灵活性，因为在系统继续运行的同时，对其进行扩展或修改变得更容易了。

本书前半部分讨论的分布式系统的原理同样可以很好地应用于基于协作的系统，尽管缓存和复制在目前的实现中所起的作用比较微弱。此外，命名与目录服务支持的基于属性的搜索有密切的联系。难以处理的是对安全性的支持，因为它实质上违反了发布者与订阅者之间的解耦合。当应该对中间件屏蔽已发布数据的内容时，进一步加剧了这个问题，使得更难以提供高效的解决方案。

习　题

1. 可以把第 4 章中讨论的消息队列系统归类为哪种协作模型?
2. 概述基于消息队列系统(如 IBM MQSeries)的发布/订阅系统的一种实现。
3. 解释为什么基于协作的分散式系统天生具有可伸缩性问题。
4. TIB/Rendezvous 中的主题名称实际上被解析为什么? 名称解析过程如何进行?
5. 概述 TIB/Rendezvous 系统中全序消息传送的一个简单实现。
6. 在像 Siena 系统(在本书中介绍过它)中使用的基于内容的路由中,我们可能遇到一个严重的管理问题。这个问题是什么?
7. 假设在 TIB/Rendezvous 系统中复制了一个进程。给出两种方法,避免来自复制进程的消息被发布多次。
8. 当在 TIB/Rendezvous 系统中复制进程时,我们在多大程度上需要全序多播?
9. 描述一种简单的 PGM 模式,允许接收方检测到有消息丢失(即使丢失的是序列中的最后一条消息)。
10. 如何在 TIB/Rendezvous 中实现一个基于生成通信的协作模型?
11. Jini 中的租约期限总是被指定为一段持续时间,而不是一个租约到期的绝对时间。为什么?
12. Jini 中最重要的可伸缩性问题是什么?
13. 考虑一个 JavaSpace 的分布式实现,其中的元组复制到多个机器上。给出一个用于删除元组的协议,以避免两个进程试图删除同一元组时的竞争情况发生。
14. 假定 Jini 中的一个事务 T 请求锁定一个当前正由另一个事务 T′锁定的对象。请解释会发生什么。
15. 假定一个 Jini 客户缓存它从一个 JavaSpace 获取的元组,以便下次不用再访问 JavaSpace。这种缓存是否有用?
16. 回答上一个问题,但这次是客户缓存查询服务返回的结果。
17. 概述容错 JavaSpace 的一种简单实现。
18. 在一些基于主题的发布/订阅系统中,安全的解决方案要求在发布者与订阅者之间进行端到端加密。不过,这种方法可能违反基于协作的系统的初始设计目标。它是如何违反的?

第14章

补充读物与参考文献

在前面的13章中，我们涉及到了很多内容。本章对有兴趣进一步研究分布式系统的读者会有所帮助。14.1节是建议阅读的材料列表，14.2节是按字母顺序排列的本书引用的书籍和文章的列表。

14.1 进一步阅读的建议

14.1.1 介绍性和综述性的著作

Coulouris 等，*Distributed Systems—Concepts and Design* 一本全面介绍分布式系统的好书。该书介绍了与本书类似的内容，但组织内容的方式截然不同。包含很多关于分布式事务的内容，但在命名和安全方面则描述得比较少。案例学习包括 CORBA 和 Mach 分布式操作系统。

Foster 与 Kesselman，*The Grid 2: Blueprint for a New Computing Infrastructure* 该书为第2版，其中很多网格专家给出了大规模网格计算的各种问题。该书涵盖了所有重要主题，包括有关当前和今后应用的示例。

Neuman，"Scale in Distributed Systems"该论文是系统地阐述分布式系统扩展性问题的为数不多的论文之一。主要作为扩展时使用的技术来探讨缓存、复制以及分布技术，并提供将这些技术应用于大型系统设计时的经验法则。

Silberschatz 等，*Applied Operating System Concepts* 这是一本全面的关于操作系统的教科书，其中包括有关分布式系统的内容，重点是文件系统和分布式协作。

Verissimo 和 Rodrigues，*Distributed Systems for Systems Architects* 这是一本有关分布式系统的高级读物，介绍的内容与本书基本相同。相对而言，该书更侧重于容错、实时分布式系统以及分布式系统的管理。

Zhao 与 Guibas，*Wireless Sensor Networks* 关于（无线）传感器网络的很多书都是以网络技术的方法来描述这些系统的。这本书则是从系统的角度来介绍的，对于那些对分布式系统感兴趣的读者来说，这本书值得一读。这本书很好地介绍了无线传感器网络。

14.1.2 体系结构

Babaoglu 等,*Self-star Properties in Complex Information Systems* 很多书说介绍的是自主系统,但并不总是令人满意。这本书包含了来自不同背景的作者的文章集,介绍了现代计算机系统中各种自主性。

Bass 等,*Software Architecture in Practice* 这是一本被广泛使用的书,全面而实用地介绍了软件体系结构。尽管这本书的重点并不是特别针对分布式系统的,但它为理解组织复杂软件系统的各种方法提供了基础。

Hellerstein 等,*Feedback Control of Computing Systems* 对有一定数学基础的读者来说,这本书详细介绍了反馈控制循环是如何应用到(分布式)计算机系统中的。因此,它是研究自主和自治系统的又一参考读物。

Lua 等,"A Survey and Comparison of Peer-to-Peer Overlay Network Schemes"是一篇关于现代点对点系统的优秀文章,涵盖了结构化和非结构化网络。对那些想深入了解该主题但不知如何下手的人来说,这篇文章是一个很好的概述。

Oram,*Peer-to-Peer: Harnessing the Power of Disruptive Technologies* 本书涵盖了有关第一代点对点网络的大量文章。该书介绍了各种项目及重要问题,如安全、托管和可计量性。尽管点对点技术已经取得了很多进展,这本书对理解很多需要解决的基本问题仍然很有价值。

White 等,"An Architectural Approach to Autonomic Computing"该文章是由技术人员写的,其背后的思想是自治计算。这篇短小的文章高度概述了满足自主系统所需的需求。

14.1.3 进程

Andrews,*Foundations of Multithreaded, Parallel, and Distributed Programming* 如果你需要一本透彻地介绍编写并行系统和分布式系统的指导书籍,那么该书正是您所需要的。

Lewis 和 Berg,*Multithreaded Programming with PthreadsPthreads* 构成了实现操作系统线程的 POSIX 标准,并受到 UNIX 系统的广泛支持。尽管作者集中介绍了 Pthreads,但书中还是提供了线程编程的精彩指南。同样,本书还包含了开发多线程客户和服务器所需的基础知识。

Schmidt 等,*Pattern-Oriented Software Architecture—Patterns for Concurrent and Networked Objects* 只是在最近,研究人员才开始注意分布式系统中的通用设计模式。这些模式可简化分布式系统的开发过程,原因是能够使程序员将精力更集中于系统特定的问题上。该书讨论了设备访问、事件处理、同步和并发的设计模式。

Smith 和 Nair,*Virtual Machines: Versatile Platforms for Systems and Processes* 这两位作者在 2005 年 5 月发表了有关虚拟技术的概要,而这本书则深入介绍了虚拟机的很多方面。正如在本书中已经介绍过的那样,虚拟机对分布式系统变得越练越重要了。这本书更多地介绍了这一主题。

Stevens 和 Rago,*Advanced Programming in the UNIX Environment* 如果有必要专门买本书来了解有关在 UNIX 系统中的编程,那么可以考虑这本书。与 Richard Stevens 编写的其他书一样,这本书包含了有关如何开放服务器和其他类型的程序的丰富信息。第 2 版由 Rago 进行了扩展。Rago 也是有关类似主题的图书的知名作者。

14.1.4 通信

Birrell 和 Nelson,"Implementing Remote Procedure Calls"这是一篇关于设计和实现最早的远程过程调用系统之一的经典论文。

Hohpe 和 Woolf,*Enterprise Integration Patterns* 与其他有关设计模式的图书一样,本书高度概述了如何构建消息传递解决方法。对那些需要设计基于消息的解决方法的读者来说,这本书非常值得一读,它介绍了很多在设计阶段可以遵循的模式。

Peterson 和 Davie,*Computer Networks, A Systems Approach* 这是有关计算机网络的又一本好书,其叙述方法与本书有些类似,先介绍一些原理,然后介绍如何把它们应用到网络中。

Steinmetz 和 Nahrstedt,*Multimedia Systems* 一本很好的教材(尽管编辑得很糟糕),介绍了有关进行多媒体处理的(分布式)系统的很多内容,对本内容也进行了很好的介绍。

14.1.5 命名

Albitz 和 Liu,*DNS and BIND* BIND 是广泛使用的一种 DNS 服务器实现,它是容易得到的公共软件。该书详尽地阐述了使用 BIND 建立 DNS 域的方法。同时还提供了当今用得最多的分布式命名服务的大量实际信息。

Balakrishnan 等,"Looking up Data in P2P Systems"这是一篇关于点对点系统中的查找机制的易读的很好文章。尽管只介绍了这些机制的实际工作原理,但为以后的阅读提供了一个很好的开端。

Balakrishnan 等,"A Layered Naming Architecture for the Internet"在这篇文章中,作者试图把结构化命名和平面命名组合在一起,从而分成了不同的三层:(1)用户友好的名称,被映射到服务标识符;(2)服务标识符,被映射到端点标识符,这些端点标识符用于唯一地标识一个主机;(3)端点,映射到网络地址。当然,对那些只使用了标识符的部分,可以很方便地使用基于 DHT 的系统。

Loshin,*Big Book of Lightweight Directory Access Protocol(LDAP)RFCs* 基于 LDAP 的系统在分布式系统中得到了迅速的普及。LDAP 服务的最终起源是由 IETF 发布的 RFC。Loshin 将相关内容汇集起来,编写出这本全面的关于 LDAP 服务的设计和实现的参考。

Needham,"Names"这是一篇关于分布式系统中名称角色的浅显易懂的精彩文章。其重点在于命名系统(请参阅本书 4.1 节),使用 DEC 的 GNS 作为示例。

Pitoura 和 Samaras,"Locating Objects in Mobile Computing"这篇文章可作为关于定位服务的全面指导。作者讨论了各种不同的定位服务,包括电信系统中使用的定位服务。

本文包含了参考资料的广泛列表，可作为进一步阅读其他文献的起始点。

Saltzer,"Naming and Binding Objects"虽然该文章写于1978年，并且关注的也不是分布式系统，但应该将该文作为命名研究的起始点。文中论述了名称与对象之间的关系，尤其是将名称解析为所引用的对象。另外也着重介绍了封闭机制的概念。

14.1.6 同步化

Guerraoui 和 Rodrigues, *Introduction to Reliable Distributed Programming* 看到这本书的书名可能产生误解，这本书主要关注的是实现可靠性的分布式算法。该书有配套的软件，可以实际测试很多理论描述。

Lynch, *Distributed Algorithms* 此书使用单一框架描述了多种不同的分布式算法。其中考虑了三种不同的定时模型：单一同步模型、无任何定时假设的异步模型以及部分接近实时系统的同步模型。如果您以前学习的是理论性的概念，那么您将发现该书包含了大量实用的算法。

Raynal 和 Singhal,"Logical Time: Capturing Causality in Distributed Systems"这篇文章用相对简单的术语描述了三种逻辑时钟：标量时间(例如 Lamport 时间戳)、矢量时间以及矩阵时间。另外，该文章章还介绍了大量实际的和实验性的分布式系统中所使用的多种实现。

Tel, *Introduction to Distributed Algorithms* 这是关于分布式算法的又一本入门教材，该书只关注消息传递系统的解决方法。尽管理论性很强，在很多实例中，读者还是可以很容易地为实际系统构建解决方法。

14.1.7 一致性与复制

Adve 和 Gharachorloo,"Shared Memory Consistency Models: A Tutorial"该文章很好地介绍了内存一致性模型。

Gray,"The Dangers of Replication and a Solution"该文章讨论的是实现顺序一致性模型的复制(称为期望复制)与懒惰复制之间的平衡。这两种复制形式都对事务处理进行了公式化。期望复制的问题是可扩展性差，而懒惰复制则可能容易导致更难以解决或可能有冲突的解决方案。作者的目的正是促成一种混合型的模式。

Saito 和 Shapiro,"Optimistic Replication"此报告展示了弱一致性模型中使用的优化复制算法的分类。报告还说明了另一种审视复制及其关联的一致性协议的方法。其中的一个有趣的话题是对各种解决方案可扩展性的讨论。该报告还包含了一个参考资料的广泛列表。

Sivasubramanian 等,"Replication for Web Hosting Systems"在这篇文章中，作者讨论在处理 Web 宿主系统的复制时需要解决的问题，包括复制替换、一致性协议和对最佳复制的路由请求。该文章还包含了一些相关材料。

Wiesmann 等,"Understanding Replication in Databases and Distributed Systems"一般来说，分布式数据库和常规的分布式系统中处理复制的方法有一些不同。在数据库中，复

制的主要原因常常是为了提高性能。而在常规分布式系统中，复制常常是为了提高容错性。该文章展示了一个能够更容易地比较解决方案的框架。

14.1.8 容错性

Marcus 和 Stern，*Blueprints for High Availability* 在开发（分布式系统）以获得高可用性时，需要考虑很多问题。本书的作者以一种注重实际的方法，介绍了很多技术和非技术问题。

Birman，*Reliable Distributed Systems* 这本书由该领域的权威人士编写，涵盖了在开放高度依赖的分布式系统时的很多陷阱信息。作者提供了很多来自学术界和工业界的示例来说明怎样做可能会犯错，如何处理。该书涵盖了大量主题，包括客户/服务器计算、Web服务、基于对象的系统（CORBA）以及点对点系统。

Cristian 和 Fetzer，"The Timed Asynchronous Distributed System Model"该文章讨论了一种更现实的分布式系统模型，而不是单纯的同步或异步模型。两个重要的假设是：服务完全在一个指定的时间间隔内发生；通信是不可靠的，并会造成性能故障。该文章阐述了这种模型在捕获实际分布式系统的重要属性时的适用性。

Guerraoui 和 Schiper，"Software-Based Replication for Fault Tolerance"该文章简要清晰地描述了分布式系统中的复制是如何增强容错性的。该文章还讨论了主机—备份复制和主动复制，以及与组通信相关的复制。

Jalote，Fault Tolerance in Distributed Systems 此书是少数几本完整地阐述分布式系统中的容错性的教材之一，内容涵盖了可靠的广播、恢复、复制和进程恢复。其中有单独的一章来解释软件设计的错误。

14.1.9 安全性

Anderson，Security Engineering：*A Guide to Building Dependable Distributed Systems* 此书是少数几本旨在阐述整个安全领域的成功教材之一，讨论了诸如口令、访问控制和密码编译等基础知识。安全性与应用程序域紧密相关，文中讨论了下面几个领域中的安全性：军事领域、银行业、医药系统及其他系统。最后还讨论了社会、组织和政治方面的安全性。该书是进一步研究安全性的最佳起始点。

Bishop，*Computer Security*：*Art and Science* 尽管本书并不是专门为分布式系统而编写的，但它含有大量有关计算机安全的常见问题的内容，包括很多在第 9 章中介绍的主题。而且，该书介绍了安全策略、安全保证、安全评估和很多实现问题。

Blaze 等，"The Role of Trust Management in Distributed Systems Security"该文章阐述的主题是大型分布式系统应该使用一种比当前所用的更简单的方法来授权对资源的访问。尤其是如果请求的证书集符合本地的安全策略，那么应该准许该请求。换句话说，即不需额外的身份验证和访问控制就应该授权。该文章解释了这种模型，并展示了其实现方式。

Kaufman，*Network Security* 这本权威的、多次再版的书最先介绍了网络安全。书中详尽地阐述了密钥和公钥算法及协议、消息散列、身份验证、Kerberos 和电子邮件。最精彩的

部分是作者之间的讨论,以下标形式标记出来,如"I2 could not get me1 to be very specific…"。

Menezes 等,*Handbook of Applied Cryptography* 书名说明了一切。该书为理解加密、散列等的不同密码解决方法提供了必需的数学背景知识。该书以单独的章节介绍了认证、数字签名、密钥创建和密钥管理等内容。

Rafaeli 和 Hutchison,"A Survey of Key Management for Secure Group Communication"文章名说明了一切。作者讨论了在进程组需要以安全分方式进行相互通信和交互的系统中可以使用的各种方案。该文章主要介绍了管理和发布密钥的各种方法。

Schneier,*Secrets and Lies* 与 Applied Cryptography 的作者为同一人,此书主要是为非技术人员解释安全性问题。一个重要的观点是:安全性不仅仅是技术问题。实际上,从本书中可以了解到大多数与安全性相关的风险可能都是人为造成的或由我们组织事务的方式造成的。此书还提供了很多材料,可作为本书第 9 章的补充读物。

14.1.10 面向对象的分布式系统

Emmerich,*Engineering Distributed Objects* 一本优秀的专门书籍,全面介绍了远程对象技术,特别关注了 CORBA、DCOM 和 Java RMI。而且此书还提供了比较这三种流行的对象模型的良好的基础知识。还包含与设计下述系统有关的材料:使用远程对象、处理不同形式的通信、定位对象、持续性、事务和安全性。

Fleury 和 Reverbel,"The JBoss Extensible Server"很多 Web 应用程序是基于 JBoss J2EE 对象服务器的。在这篇文章中,服务器的初始开发者给出了基本原理和一般设计。

Henning,"The Rise and Fall of CORBA"该文章是由 CORBA 开发方面的专家所写的,作者强烈地反对 CORBA 的使用。最为突出的是,作者认为,CORBA 太复杂,它并不能使分布式系统开发人员活得更轻松。

Henning 和 Vinoski,*Advanced CORBA Programming with C++* 如果需要 CORBA 编程的资料,并且同时需要学习大量关于 CORBA 实质的内容,那么此书正是最佳选择。此书是由参与了 CORBA 系统的设计与开发的技术人员编写的,此书的实际信息和技术细节性信息并不局限于某个特定的 CORBA 实现。

14.1.11 分布式文件系统

Blanco 等,"A Survey of Data Management in Peer-to-Peer Systems"这是一篇内容广泛的文章,介绍了很多重要的点对点系统。作者介绍了数据管理问题,包括数据集成、查询处理和数据一致性等。

Pate,*UNIX Filesystems: Evolution, Design, and Implementation* 该书介绍了很多为 UNIX 系统开发的文件系统,而且还有专门一章介绍分布式文件系统。

Satyanarayanan,"The Evolution of Coda"Coda 是用于支持移动用户的一种重要的分布式文件系统。尤其是,它具有支持断开操作的特性,这样,用户就可以在与主服务器断开的情况下,继续对自己的文件进行操作。该文章介绍了多年来,随着新需求的出现,文

件系统是如何发展的。

Zhu 等，"Hibernator：Helping Disk Arrays Sleep through the Winter"数据中心使用了非常大量的磁盘来完成它们的工作。显然，这需要大量的能源。该文章介绍了把能耗降下来的各种技术，例如，把热点数据与不常访问的数据分开等。

14.1.12 基于 Web 的分布式系统

Alonso 等，*Web Services：Concepts，Architectures and Applications* Web 服务的普及和复杂使得有关它的文章大量出现，但很多只能看作是垃圾。而这书是清晰介绍 Web 服务的少数几本之一。向初学者强烈推荐。

Chappel，*Understanding .NET* 微软公司用于支持 Web 服务开发的方法是，把它们已有的很多技术组合到一个框架中，并加入了很多新东西。其结果就是 .NET。这种方法使得人们对这种框架实际是什么产生了困惑。该书作者很好地解释了这件事情。

Fielding，"Principled Design of the Modern Web Architecture"作者是 Apache Web 服务器的主要设计人员。该文章讨论了组织 Web 应用程序的一般方法，以便它们可以最佳地使用当前的协议集。

Podling 和 Boszormenyi，"A Survey of Web Cache Replacement Strategies"我们很少触及这个工作：当 Web 高速缓存满时需要做什么。该文章很好地概述了当高速缓充满时，从中取回内容的各种方法。

Rabinovich 和 Spatscheck，*Web Caching and Replication* 这是一本很好的书，全面并深入地介绍了 Web 中的内容分布性问题。

Sebesta，*Programming the World Wide Web* 我们很少触及实际的 Web 应用程序开发。Web 应用程序的开发往往涉及各种不同的工具和技术。本书进行了全面的介绍，是学习 Web 站点开发的很好起点。

14.1.13 基于协作的分布式系统

Cabri 等，"Uncoupling Coordination：Tuple-based Models for Mobility"该文章作者很好地介绍了类 Linda 系统，这种系统可以在移动的、分布式环境下操作。该文章还显示，15 年以前开始的一些领域现在仍在研究之中。

Pietzuch 和 Bacon，"Hermes：A Distrib. Event-Based Middleware Architecture"Hermes 是一个分布式的发布/订阅系统，由英国剑桥大学开发。它已用作很多基于事件的大型系统的基础。该文章描述了 Hermes 的基本组织结构。

Wells 等，"Linda Implementation in Java for Concurrent Systems"那些对用 Java 语言的元组空间实现感兴趣的读者，该文章提供了很好的概述。它更多的是关注计算而不是一般的元组空间应用，并阐述了当性能是关键时，需要进行的一些折衷权衡。

Zhao 等，"Subscription Propagation in Highly-Available Publish/Subscribe Middleware"尽管是一篇技术性很强的文章，该文章还是给出了一些问题的很好解决方法，当可用性在发布/订购系统中是一个重要的设计标准时，起着很重要的作用。

14.2 参考文献

ABADI, M. and NEEDHAM, R.: "Prudent Engineering Practice for Cryptographic Protocols." *IEEE Trans. Softw. Eng.*, (22)1:6-15, Jan. 1996.

ABDULLAHI, S. and RINGWOOD, G.: "Garbage Collecting the Internet: A Survey of Distributed Garbage Collection." *ACM Comput. Surv.*, (30)3:330-373, Sept. 1998.

ABERER, K. and HAUSWIRTH, M.: "Peer-to-Peer Systems." In Singh, M. (ed.), *The Practical Handbook of Internet Computing*, chapter 35. Boca Raton, FL: CRC Press, 2005.

ABERER, K., ALIMA, L. O., GHODSI, A., GIRDZIJAUSKAS, S., HAUSWIRTH, M., and HARIDI, S.: "The Essence of P2P: A Reference Architecture for Overlay Networks." *Proc. Fifth Int'l Conf. Peer-to-Peer Comput.*, (Konstanz, Germany). Los Alamitos, CA: IEEE Computer Society Press, 2005. pp. 11-20.

ADAR, E. and HUBERMAN, B. A.: "Free Riding on Gnutella." Hewlett Packard, Information Dynamics Lab, Jan. 2000.

AIYER, A., ALVISI, L., CLEMENT, A., DAHLIN, M., and MARTIN, J.-P.: "BAR Fault Tolerance for Cooperative Services." *Proc. 20th Symp. Operating System Principles*, (Brighton, UK). New York, NY: ACM Press, 2005. pp. 45-58.

AKYILDIZ, I. F., SU, W., SANKARASUBRAMANIAM, Y., and CAYIRCI, E.: "A Survey on Sensor Networks." *IEEE Commun. Mag.*, (40)8:102-114, Aug. 2002.

AKYILDIZ, I. F., WANG, X., and WANG, W.: "Wireless Mesh Networks: A Survey." *Comp. Netw.*, (47)4:445-487, Mar. 2005.

ALBITZ, P. and LIU, C.: *DNS and BIND*. Sebastopol, CA: O'Reilly & Associates, 4th ed., 2001.

ALLEN, R. and LOWE-NORRIS, A.: *Windows 2000 Active Directory*. Sebastopol, CA: O'Reilly & Associates, 2nd ed., 2003.

ALLMAN, M.: "An Evaluation of XML-RPC." *Perf. Eval. Rev.*, (30)4:2-11, Mar. 2003.

ALONSO, G., CASATI, F., KUNO, H., and MACHIRAJU, V.: *Web Services: Concepts, Architectures and Applications*. Berlin: Springer-Verlag, 2004.

ALVISI, L. and MARZULLO, K.: "Message Logging: Pessimistic, Optimistic, Causal, and Optimal." *IEEE Trans. Softw. Eng.*, (24)2:149-159, Feb. 1998.

AMAR, L., BARAK, A., and SHILOH, A.: "The MOSIX Direct File System Access Method for Supporting Scalable Cluster File Systems." *Cluster Comput.*, (7)2:141-150, Apr. 2004.

ANDERSON, O. T., LUAN, L., EVERHART, C., PEREIRA, M., SARKAR, R., and XU, J.: "Global Namespace for Files." *IBM Syst. J.*, (43)4:702-722, Apr. 2004.

ANDERSON, R.: *Security Engineering – A Guide to Building Dependable Distributed Systems*. New York: John Wiley, 2001.

ANDERSON, T., BERSHAD, B., LAZOWSKA, E., and LEVY, H.: "Scheduler Activations:

Efficient Kernel Support for the User-Level Management of Parallelism." *Proc. 13th Symp. Operating System Principles*. New York, NY: ACM Press, 1991. pp. 95-109.

ANDREWS, G.: *Foundations of Multithreaded, Parallel, and Distributed Programming*. Reading, MA: Addison-Wesley, 2000.

ANDROUTSELLIS-THEOTOKIS, S. and SPINELLIS, D.: "A Survey of Peer-to-Peer Content Distribution Technologies." *ACM Comput. Surv.*, (36)4:335-371, Dec. 2004.

ARAUJO, F. and RODRIGUES, L.: "Survey on Position-Based Routing." Technical Report MINEMA TR-01, University of Lisbon, Oct. 2005.

ARKILLS, B.: *LDAP Directories Explained: An Introduction and Analysis*. Reading, MA: Addison-Wesley, 2003.

ARON, M., SANDERS, D., DRUSCHEL, P., and ZWAENEPOEL, W.: "Scalable Content-aware Request Distribution in Cluster-based Network Servers." *Proc. USENIX Ann. Techn. Conf.* USENIX, 2000. pp. 323-336.

ATTIYA, H. and WELCH, J.: *Distributed Computing Fundamentals, Simulations, and Advanced Topics*. New York: John Wiley, 2nd ed., 2004.

AVIZIENIS, A., LAPRIE, J.-C., RANDELL, B., and LANDWEHR, C.: "Basic Concepts and Taxonomy of Dependable and Secure Computing." *IEEE Trans. Depend. Secure Comput.*, (1)1:11-33, Jan. 2004.

AWADALLAH, A. and ROSENBLUM, M.: "The vMatrix: A Network of Virtual Machine Monitors for Dynamic Content Distribution." *Proc. Seventh Web Caching Workshop*, (Boulder, CO), 2002.

AWADALLAH, A. and ROSENBLUM, M.: "The vMatrix: Server Switching." *Proc. Tenth Workshop on Future Trends in Distributed Computing Systems*, (Suzhou, China). Los Alamitos, CA: IEEE Computer Society Press, 2004. pp. 110-118.

BABAOGLU, O., JELASITY, M., MONTRESOR, A., FETZER, C., LEONARDI, S., VAN MOORSEL, A., and VAN STEEN, M. (eds.): *Self-star Properties in Complex Information Systems*, vol. 3460 of *Lect. Notes Comp. Sc.* Berlin: Springer-Verlag, 2005.

BABAOGLU, O. and TOUEG, S.: "Non-Blocking Atomic Commitment." In Mullender, S. (ed.), *Distributed Systems*, pp. 147-168. Wokingham: Addison-Wesley, 2nd ed., 1993.

BABCOCK, B., BABU, S., DATAR, M., MOTWANI, R., and WIDOM, J.: "Models and Issues in Data Stream Systems." *Proc. 21st Symp. on Principles of Distributed Computing*, (Monterey, CA). New York, NY: ACM Press, 2002. pp. 1-16.

BAL, H.: *The Shared Data-Object Model as a Paradigm for Programming Distributed Systems*. Ph.D.. Thesis, Vrije Universiteit, Amsterdam, 1989.

BALAKRISHNAN, H., KAASHOEK, M. F., KARGER, D., MORRIS, R., and STOICA, I.: "Looking up Data in P2P Systems." *Commun. ACM*, (46)2:43-48, Feb. 2003.

BALAKRISHNAN, H., LAKSHMINARAYANAN, K., RATNASAMY, S., SHENKER, S., STOICA, I., and WALFISH, M.: "A Layered Naming Architecture for the Internet." *Proc. SIGCOMM*, (Portland, OR). New York, NY: ACM Press, 2004. pp. 343-352.

BALAZINSKA, M., BALAKRISHNAN, H., and KARGER, D.: "INS/Twine: A Scalable Peer-to-Peer Architecture for Intentional Resource Discovery." *Proc. First Int'l Conf. Pervasive Computing*, vol. 2414 of *Lect. Notes Comp. Sc.*, (Zurich, Switzerland). Berlin: Springer-Verlag, 2002. pp. 195-210.

BALLINTIJN, G.: *Locating Objects in a Wide-area System.* Ph.D. thesis, Vrije Universiteit Amsterdam, 2003.

BARATTO, R. A., NIEH, J., and KIM, L.: "THINC: A Remote Display Architecture for Thin-Client Computing." *Proc. 20th Symp. Operating System Principles*, (Brighton, UK). New York, NY: ACM Press, 2005. pp. 277-290.

BARBORAK, M., MALEK, M., and DAHBURA, A.: "The Consensus Problem in Fault-Tolerant Computing." *ACM Comput. Surv.*, (25)2:171-220, June 1993.

BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAR, R., PRATT, I., and WARFIELD, A.: "Xen and the Art of Virtualization." *Proc. 19th Symp. Operating System Principles*, (Bolton Landing, NY). New York, NY: ACM Press, 2003. pp. 164-177.

BARKER, W.: "Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher." NIST Special Publication 800-67, May 2004.

BARRON, D.: *Pascal – The Language and its Implementation.* New York: John Wiley, 1981.

BARROSO, L., DEAM, J., and HOLZE, U.: "Web Search for a Planet: The Google Cluster Architecture." *IEEE Micro*, (23)2:21-28, Mar. 2003.

BARYSHNIKOV, Y., COFFMAN, E. G., PIERRE, G., RUBENSTEIN, D., SQUILLANTE, M., and YIMWADSANA, T.: "Predictability of Web-Server Traffic Congestion." *Proc. Tenth Web Caching Workshop*, (Sophia Antipolis, France). IEEE, 2005. pp. 97-103.

BASILE, C., KALBARCZYK, Z., and IYER, R. K.: "A Preemptive Deterministic Scheduling Algorithm for Multithreaded Replicas." *Proc. Int'l Conf. Dependable Systems and Networks*, (San Francisco, CA). Los Alamitos, CA: IEEE Computer Society Press, 2003. pp. 149-158.

BASILE, C., WHISNANT, K., KALBARCZYK, Z., and IYER, R. K.: "Loose Synchronization of Multithreaded Replicas." *Proc. 21st Symp. on Reliable Distributed Systems*, (Osaka, Japan). Los Alamitos, CA: IEEE Computer Society Press, 2002. pp. 250-255.

BASS, L., CLEMENTS, P., and KAZMAN, R.: *Software Architecture in Practice.* Reading, MA: Addison-Wesley, 2nd ed., 2003.

BAVIER, A., BOWMAN, M., CHUN, B., CULLER, D., KARLIN, S., MUIR, S., PETERSON, L., ROSCOE, T., SPALINK, T., and WAWRZONIAK, M.: "Operating System Support for Planetary-Scale Network Services." *Proc. First Symp. Networked Systems Design and Impl.*, (San Francisco, CA). Berkeley, CA: USENIX, 2004. pp. 245-266.

BERNERS-LEE, T., CAILLIAU, R., NIELSON, H. F., and SECRET, A.: "The World-Wide Web." *Commun. ACM*, (37)8:76-82, Aug. 1994.

BERNERS-LEE, T., FIELDING, R., and MASINTER, L.: "Uniform Resource Identifiers (URI): Generic Syntax." RFC 3986, Jan. 2005.

BERNSTEIN, P.: "Middleware: A Model for Distributed System Services." *Commun. ACM*, (39)2:87-98, Feb. 1996.

BERNSTEIN, P., HADZILACOS, V., and GOODMAN, N.: *Concurrency Control and Recovery in Database Systems.* Reading, MA: Addison-Wesley, 1987.

BERSHAD, B., ZEKAUSKAS, M., and SAWDON, W.: "The Midway Distributed Shared Memory System." *Proc. COMPCON.* IEEE, 1993. pp. 528-537.

BHAGWAN, R., TATI, K., CHENG, Y., SAVAGE, S., and VOELKER, G. M.: "Total Recall: Systems Support for Automated Availability Management." *Proc. First Symp. Networked Systems Design and Impl.*, (San Francisco, CA). Berkeley, CA: USENIX, 2004. pp. 337-350.

BHARAMBE, A. R., AGRAWAL, M., and SESHAN, S.: "Mercury: Supporting Scalable Multi-Attribute Range Queries." *Proc. SIGCOMM*, (Portland, OR). New York, NY: ACM Press, 2004. pp. 353-366.

BIRMAN, K.: *Reliable Distributed Systems: Technologies, Web Services, and Applications.* Berlin: Springer-Verlag, 2005.

BIRMAN, K.: "A Response to Cheriton and Skeen's Criticism of Causal and Totally Ordered Communication." *Oper. Syst. Rev.*, (28)1:11-21, Jan. 1994.

BIRMAN, K. and JOSEPH, T.: "Reliable Communication in the Presence of Failures." *ACM Trans. Comp. Syst.*, (5)1:47-76, Feb. 1987.

BIRMAN, K., SCHIPER, A., and STEPHENSON, P.: "Lightweight Causal and Atomic Group Multicast." *ACM Trans. Comp. Syst.*, (9)3:272-314, Aug. 1991.

BIRMAN, K. and VAN RENESSE, R. (eds.): *Reliable Distributed Computing with the Isis Toolkit.* Los Alamitos, CA: IEEE Computer Society Press, 1994.

BIRRELL, A. and NELSON, B.: "Implementing Remote Procedure Calls." *ACM Trans. Comp. Syst.*, (2)1:39-59, Feb. 1984.

BISHOP, M.: *Computer Security: Art and Science.* Reading, MA: Addison-Wesley, 2003.

BJORNSON, R.: *Linda on Distributed Memory Multicomputers.* Ph.D. Thesis, Yale University, Department of Computer Science, 1993.

BLACK, A. and ARTSY, Y.: "Implementing Location Independent Invocation." *IEEE Trans. Par. Distr. Syst.*, (1)1:107-119, Jan. 1990.

BLAIR, G., COULSON, G., and GRACE, P.: "Research Directions in Reflective Middleware: the Lancaster Experience." *Proc. Third Workshop Reflective & Adaptive Middleware*, (Toronto, Canada). New York, NY: ACM Press, 2004. pp. 262-267.

BLAIR, G. and STEFANI, J.-B.: *Open Distributed Processing and Multimedia.* Reading, MA: Addison-Wesley, 1998.

BLAKE-WILSON, S., NYSTROM, M., HOPWOOD, D., MIKKELSEN, J., and WRIGHT, T.: "Transport Layer Security (TLS) Extensions." RFC 3546, June 2003. Cited on page 584.

BLANCO, R., AHMED, N., HADALLER, D., SUNG, L. G. A., LI, H., and SOLIMAN, M. A.: "A Survey of Data Management in Peer-to-Peer Systems." Technical Report CS-2006-18, University of Waterloo, Canada, June 2006.

BLAZE, M., FEIGENBAUM, J., IOANNIDIS, J., and KEROMYTIS, A.: "The Role of Trust Management in Distributed Systems Security." In Vitek, J. and Jensen, C. (eds.), *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, vol. 1603 of *Lect. Notes Comp. Sc.*, pp. 185-210. Berlin: Springer-Verlag, 1999.

BLAZE, M.: *Caching in Large-Scale Distributed File Systems.* Ph.D. thesis, Department of Computer Science, Princeton University, Jan. 1993.

BONNET, P., GEHRKE, J., and SESHADRI, P.: "Towards Sensor Database Systems." *Proc. Second Int'l Conf. Mobile Data Mgt.*, vol. 1987 of *Lect. Notes Comp. Sc.*, (Hong Kong, China). Berlin: Springer-Verlag, 2002. pp. 3-14.

BOOTH, D., HAAS, H., MCCABE, F., NEWCOMER, E., CHAMPION, M., FERRIS, C., and ORCHARD, D.: "Web Services Architecture." W3C Working Group Note, Feb. 2004.

BOUCHENAK, S., BOYER, F., HAGIMONT, D., KRAKOWIAK, S., MOS, A., DE PALMA3, N., QUEMA3, V., and STEFANI, J.-B.: "Architecture-Based Autonomous Repair Management: An Application to J2EE Clusters." *Proc. 24th Symp. on Reliable Distributed Systems*, (Orlando, FL). Los Alamitos, CA: IEEE Computer Society Press, 2005. pp. 13-24.

BREWER, E.: "Lessons from Giant-Scale Services." *IEEE Internet Comput.*, (5)4:46-55, July 2001.

BRUNETON, E., COUPAYE, T., LECLERCQ, M., QUEMA, V., and STEFANI, J.-B.: "An Open Component Model and Its Support in Java." *Proc. Seventh Int'l Symp. Component-based Softw. Eng.*, vol. 3054 of *Lect. Notes Comp. Sc.*, (Edinburgh, UK). Berlin: Springer-Verlag, 2004. pp. 7-22.

BUDHIJARA, N., MARZULLO, K., SCHNEIDER, F., and TOUEG, S.: "The Primary-Backup Approach." In Mullender, S. (ed.), *Distributed Systems*, pp. 199-216. Wokingham: Addison-Wesley, 2nd ed., 1993.

BUDHIRAJA, N. and MARZULLO, K.: "Tradeoffs in Implementing Primary-Backup Protocols." Technical Report TR 92-1307, Department of Computer Science, Cornell University, 1992.

BURNS, R. C., REES, R. M., STOCKMEYER, L. J., and LONG, D. D. E.: "Scalable Session Locking for a Distributed File System." *Cluster Computing*, (4)4:295-306, Oct. 2001.

BUSI, N., MONTRESOR, A., and ZAVATTARO, G.: "Data-driven Coordination in Peer-to-Peer Information Systems." *Int'l J. Coop. Inf. Syst.*, (13)1:63-89, Mar. 2004.

BUTT, A. R., JOHNSON, T. A., ZHENG, Y., and HU, Y. C.: "Kosha: A Peer-to-Peer Enhancement for the Network File System." *Proc. Int'l Conf. Supercomputing*, (Washington, DC). Los Alamitos, CA: IEEE Computer Society Press, 2004. pp. 51-61.

CABRI, G., FERRARI, L., LEONARDI, L., MAMEI, M., and ZAMBONELLI, F.: "Uncoupling Coordination: Tuple-based Models for Mobility." In Bellavista, Paolo and Corradi, Antonio (eds.), *The Handbook of Mobile Middleware*. London, UK: CRC Press, 2006.

CABRI, G., LEONARDI, L., and ZAMBONELLI, F.: "Mobile-Agent Coordination Models for Internet Applications." *IEEE Computer*, (33)2:82-89, Feb. 2000.

CAI, M., CHERVENAK, A., and FRANK, M.: "A Peer-to-Peer Replica Location Service Based on A Distributed Hash Table." *Proc. High Perf. Comput., Netw., & Storage Conf.*, (Pittsburgh, PA). New York, NY: ACM Press, 2004. pp. 56-67.

CALLAGHAN, B.: *NFS Illustrated*. Reading, MA: Addison-Wesley, 2000.

CANDEA, G., BROWN, A. B., FOX, A., and PATTERSON, D.: "Recovery-Oriented Computing: Building Multitier Dependability." *IEEE Computer*, (37)11:60-67, Nov. 2004a.

CANDEA, G., KAWAMOTO, S., FUJIKI, Y., FRIEDMAN, G., and FOX, A.: "Microreboot: A Technique for Cheap Recovery." *Proc. Sixth Symp. on Operating System Design and Implementation*, (San Francisco, CA). Berkeley, CA: USENIX, 2004b. pp. 31-44.

CANDEA, G., KICIMAN, E., KAWAMOTO, S., and FOX, A.: "Autonomous Recovery in Componentized Internet Applications." *Cluster Comput.*, (9)2:175-190, Feb. 2006.

CANTIN, J., LIPASTI, M., and SMITH, J.: "The Complexity of Verifying Memory Coherence and Consistency." *IEEE Trans. Par. Distr. Syst.*, (16)7:663-671, July 2005.

CAO, L. and OSZU, T.: "Evaluation of Strong Consistency Web Caching Techniques." *World Wide Web*, (5)2:95-123, June 2002.

CAO, P. and LIU, C.: "Maintaining Strong Cache Consistency in the World Wide Web." *IEEE Trans. Comp.*, (47)4:445-457, Apr. 1998.

CAPORUSCIO, M., CARZANIGA, A., and WOLF, A. L.: "Design and Evaluation of a Support Service for Mobile, Wireless Publish/Subscribe Applications." *IEEE Trans. Softw. Eng.*, (29)12:1059-1071, Dec. 2003.

CARDELLINI, V., CASALICCHIO, E., COLAJANNI, M., and YU, P.: "The State of the Art in Locally Distributed Web-Server Systems." *ACM Comput. Surv.*, (34)2:263-311, June 2002.

CARRIERO, N. and GELERNTER, D.: "The S/Net's Linda Kernel." *ACM Trans. Comp. Syst.*, (32)2:110-129, May 1986.

CARZANIGA, A., RUTHERFORD, M. J., and WOLF, A. L.: "A Routing Scheme for Content-Based Networking." *Proc. 23rd INFOCOM Conf.*, (Hong Kong, China). Los Alamitos, CA: IEEE Computer Society Press, 2004.

CARZANIGA, A. and WOLF, A. L.: "Forwarding in a Content-based Network." *Proc. SIGCOMM*, (Karlsruhe, Germany). New York, NY: ACM Press, 2003. pp. 163-174.

CASTRO, M., DRUSCHEL, P., GANESH, A., ROWSTRON, A., and WALLACH, D. S.: "Secure Routing for Structured Peer-to-Peer Overlay Networks." *Proc. Fifth Symp. on Operating System Design and Implementation*, (Boston, MA). New York, NY: ACM Press, 2002a. pp. 299-314.

CASTRO, M., DRUSCHEL, P., HU, Y. C., and ROWSTRON, A.: "Topology-aware Routing in Structured Peer-to-Peer Overlay Networks." Technical Report MSR-TR-2002-82, Microsoft Research, Cambridge, UK, June 2002b.

CASTRO, M., RODRIGUES, R., and LISKOV, B.: "BASE: Using Abstraction to Improve Fault Tolerance." *ACM Trans. Comp. Syst.*, (21)3:236-269, Aug. 2003.

CASTRO, M., COSTA, M., and ROWSTRON, A.: "Debunking Some Myths about Structured and Unstructured Overlays." *Proc. Second Symp. Networked Systems Design and Impl.*, (Boston, MA). Berkeley, CA: USENIX, 2005.

CASTRO, M., DRUSCHEL, P., KERMARREC, A.-M., and ROWSTRON, A.: "Scribe: A Large-Scale and Decentralized Application-Level Multicast Infrastructure." *IEEE J. Selected Areas Commun.*, (20)8:100-110, Oct. 2002.

CASTRO, M. and LISKOV, B.: "Practical Byzantine Fault Tolerance and Proactive Recovery." *ACM Trans. Comp. Syst.*, (20)4:398-461, Nov. 2002.

CHAPPELL, D.: *Understanding .NET*. Reading, MA: Addison-Wesley, 2002.

CHERITON, D. and MANN, T.: "Decentralizing a Global Naming Service for Improved Performance and Fault Tolerance." *ACM Trans. Comp. Syst.*, (7)2:147-183, May 1989.

CHERITON, D. and SKEEN, D.: "Understanding the Limitations of Causally and Totally Ordered Communication." *Proc. 14th Symp. Operating System Principles*. ACM, 1993. pp. 44-57.

CHERVENAK, A., SCHULER, R., KESSELMAN, C., KORANDA, S., and MOE, B.: "Wide Area Data Replication for Scientific Collaborations." *Proc. Sixth Int'l Workshop on Grid Computing*, (Seattle, WA). New York, NY: ACM Press, 2005.

CHERVENAK, A., FOSTER, I., KESSELMAN, C., SALISBURY, C., and TUECKE, S.: "The Data Grid: Towards an Architecture for the Distributed Management and Analysis of Large Scientific Datasets." *J. Netw. Comp. App.*, (23)3:187-200, July 2000.

CHESWICK, W. and BELLOVIN, S.: *Firewalls and Internet Security*. Reading, MA: Addison-Wesley, 2nd ed., 2000.

CHOW, R. and JOHNSON, T.: *Distributed Operating Systems and Algorithms*. Reading, MA: Addison-Wesley, 1997.

CHUN, B. and SPALINK, T.: "Slice Creation and Management." Technical Report PDN-03-013, PlanetLab Consortium, July 2003.

CIANCARINI, P., TOLKSDORF, R., VITALI, F., and KNOCHE, A.: "Coordinating Multiagent Applications on the WWW: A Reference Architecture." *IEEE Trans. Softw. Eng.*, (24)5:362-375, May 1998.

CLARK, C., FRASER, K., HAND, S., HANSEN, J. G., JUL, E., LIMPACH, C., PRATT, I., and WARFIELD, A.: "Live Migration of Virtual Machines." *Proc. Second Symp. Networked Systems Design and Impl.*, (Boston, MA). Berkeley, CA: USENIX, 2005.

CLARK, D.: "The Design Philosophy of the DARPA Internet Protocols." *Proc. SIGCOMM*, (Austin, TX). New York, NY: ACM Press, 1989. pp. 106-114.

CLEMENT, L., HATELY, A., VON RIEGEN, C., and ROGERS, T.: "Universal Description, Discovery and Integration (UDDI)." Technical Report, OASIS UDDI, 2004.

COHEN, B.: "Incentives Build Robustness in Bittorrent." *Proc. First Workshop on Economics of Peer-to-Peer Systems*, (Berkeley, CA), 2003.

COHEN, D.: "On Holy Wars and a Plea for Peace." *IEEE Computer*, (14)10:48-54, Oct. 1981.

COHEN, E. and SHENKER, S.: "Replication Strategies in Unstructured Peer-to-Peer Networks." *Proc. SIGCOMM*, (Pittsburgh, PA). New York, NY: ACM Press, 2002. pp. 177-190.

COMER, D.: *Internetworking with TCP/IP, Volume I: Principles, Protocols, and Architecture*. Upper Saddle River, NJ: Prentice Hall, 5th ed., 2006.

CONTI, M., GREGORI, E., and LAPENNA, W.: "Content Delivery Policies in ReplicatedWeb Services: Client-Side vs. Server-Side." *Cluster Comput.*, (8)47-60, Jan. 2005.

COPPERSMITH, D.: "The Data Encryption Standard (DES) and its Strength Against Attacks." *IBM J. Research and Development*, (38)3:243-250, May 1994.

COULOURIS, G., DOLLIMORE, J., and KINDBERG, T.: *Distributed Systems, Concepts and Design*. Reading, MA: Addison-Wesley, 4th ed., 2005.

COX, L. and NOBLE, B.: "Samsara: Honor Among Thieves in Peer-to-Peer Storage." *Proc. 19th Symp. Operating System Principles*, (Bolton Landing, NY). New York, NY: ACM Press, 2003. pp. 120-131.

COYLER, A., BLAIR, G., and RASHID, A.: "Managing Complexity In Middleware." *Proc.*

Second AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software, 2003.

CRESPO, A. and GARCIA-MOILINA, H.: "Semantic Overlay Networks for P2P Systems." Technical Report, Stanford University, Department of Computer Science, 2003.

CRISTIAN, F.: "Probabilistic Clock Synchronization." *Distributed Computing*, (3)146-158, 1989.

CRISTIAN, F.: "Understanding Fault-Tolerant Distributed Systems." *Commun. ACM*, (34)2:56-78, Feb. 1991.

CRISTIAN, F. and FETZER, C.: "The Timed Asynchronous Distributed System Model." *IEEE Trans. Par. Distr. Syst.*, (10)6:642-657, June 1999.

CROWLEY, C.: *Operating Systems, A Design-Oriented Approach.* Chicago: Irwin, 1997.

DABEK, F., COX, R., KAASHOEK, F., and MORRIS, R.: "Vivaldi: A Decentralized Network Coordinate System." *Proc. SIGCOMM*, (Portland, OR). New York, NY: ACM Press, 2004a.

DABEK, F., KAASHOEK, M. F., KARGER, D., MORRIS, R., and STOICA, I.: "Wide-area Cooperative Storage with CFS." *Proc. 18th Symp. Operating System Principles*. ACM, 2001.

DABEK, F., LI, J., SIT, E., ROBERTSON, J., KAASHOEK, M. F., and MORRIS, R.: "Designing a dht for low latency and high throughput." *Proc. First Symp. Networked Systems Design and Impl.*, (San Francisco, CA). Berkeley, CA: USENIX, 2004b. pp. 85-98.

DAIGLE, L., VAN GULIK, D., IANNELLA, R., and FALTSTROM, P.: "Uniform Resource Names (URN) Namespace Definition Mechanisms." RFC 3406, Oct. 2002.

DAVIE, B., CHARNY, A., BENNET, J., BENSON, K., BOUDEC, J. L., COURTNEY, W., S.DAVARI, FIROIU, V., and STILIADIS, D.: "An Expedited Forwarding PHB (Per-Hop Behavior)." RFC 3246, Mar. 2002.

DAY, J. and ZIMMERMAN, H.: "The OSI Reference Model." *Proceedings of the IEEE*, (71)12:1334-1340, Dec. 1983.

DEERING, S., ESTRIN, D., FARINACCI, D., JACOBSON, V., LIU, C.-G., and WEI, L.: "The PIM Architecture for Wide-Area Multicast Routing." *IEEE/ACM Trans. Netw.*, (4)2:153-162, Apr. 1996.

DEERING, S. and CHERITON, D.: "Multicast Routing in Datagram Internetworks and Extended LANs." *ACM Trans. Comp. Syst.*, (8)2:85-110, May 1990.

DEMERS, A., GEHRKE, J., HONG, M., RIEDEWALD, M., and WHITE, W.: "Towards Expressive Publish/Subscribe Systems." *Proc. Tenth Int'l Conf. on Extended Database Technology*, (Munich, Germany), 2006.

DEMERS, A., GREENE, D., HAUSER, C., IRISH, W., LARSON, J., SHENKER, S., STURGIS, H., SWINEHART, D., and TERRY, D.: "Epidemic Algorithms for Replicated Database Maintenance." *Proc. Sixth Symp. on Principles of Distributed Computing*, (Vancouver). ACM, 1987. pp. 1-12.

DEUTSCH, P., SCHOULTZ, R., FALTSTROM, P., and WEIDER, C.: "Architecture of the WHOIS++ Service." RFC 1835, Aug. 1995.

D\'FAGO, X., SHIPER, A., and URB{'A}N, P.: "Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey." *ACM Comput. Surv.*, (36)4:372-421, Dec. 2004.

DIAO, Y., HELLERSTEIN, J., PAREKH, S., GRIFFITH, R., KAISER, G., and PHUNG, D.: "A Control Theory Foundation for Self-Managing Computing Systems." *IEEE J. Selected Areas Commun.*, (23)12:2213-2222, Dec. 2005.

DIERKS, T. and ALLEN, C.: "The Transport Layer Security Protocol." RFC 2246, Jan. 1996.

DIFFIE, W. and HELLMAN, M.: "New Directions in Cryptography." *IEEE Trans. Information Theory*, (IT-22)6:644-654, Nov. 1976.

DILLEY, J., MAGGS, B., PARIKH, J., PROKOP, H., SITARAMAN, R., and WEIHL, B.: "Globally Distributed Content Delivery." *IEEE Internet Comput.*, (6)5:50-58, Sept. 2002.

DIOT, C., LEVINE, B., LYLES, B., KASSEM, H., and BALENSIEFEN, D.: "Deployment Issues for the IP Multicast Service and Architecture." *IEEE Network*, (14)1:78-88, Jan. 2000.

DOORN, J. H. and RIVERO, L. C. (eds.): *Database Integrity: Challenges and Solutions*. Hershey, PA: Idea Group, 2002.

DOUCEUR, J. R.: "The Sybil Attack." *Proc. First Int'l Workshop on Peer-to-Peer Systems*, vol. 2429 of *Lect. Notes Comp. Sc.* Berlin: Springer-Verlag, 2002. pp. 251-260.

DUBOIS, M., SCHEURICH, C., and BRIGGS, F.: "Synchronization, Coherence, and Event Ordering in Multiprocessors." *IEEE Computer*, (21)2:9-21, Feb. 1988.

DUNAGAN, J., HARVEY, N. J. A., JONES, M. B., KOSTIC, D., THEIMER, M., and WOLMAN, A.: "FUSE: Lightweight Guaranteed Distributed Failure Notification." *Proc. Sixth Symp. on Operating System Design and Implementation*, (San Francisco, CA). Berkeley, CA: USENIX, 2004.

DUVVURI, V., SHENOY, P., and TEWARI, R.: "Adaptive Leases: A Strong Consistency Mechanism for the World Wide Web." *IEEE Trans. Know. Data Eng.*, (15)5:1266-1276, Sept. 2003.

EDDON, G. and EDDON, H.: *Inside Distributed COM*. Redmond, WA: Microsoft Press, 1998.

EISLER, M.: "LIPKEY - A Low Infrastructure Public Key Mechanism Using SPKM." RFC 2847, June 2000.

EISLER, M., CHIU, A., and LING, L.: "RPCSEC_GSS Protocol Specification." RFC 2203, Sept. 1997.

ELNOZAHY, E. N. and PLANK, J. S.: "Checkpointing for Peta-Scale Systems: A Look into the Future of Practical Rollback-Recovery." *IEEE Trans. Depend. Secure Comput.*, (1)2:97-108, Apr. 2004.

ELNOZAHY, E., ALVISI, L., WANG, Y.-M., and JOHNSON, D.: "A Survey of Rollback-Recovery Protocols in Message-Passing Systems." *ACM Comput. Surv.*, (34)3:375-408, Sept. 2002.

ELSON, J., GIROD, L., and ESTRIN, D.: "Fine-Grained Network Time Synchronization using Reference Broadcasts." *Proc. Fifth Symp. on Operating System Design and Implementation*, (Boston, MA). New York, NY: ACM Press, 2002. pp. 147-163.

EMMERICH, W.: *Engineering Distributed Objects*. New York: John Wiley, 2000.

EUGSTER, P., FELBER, P., GUERRAOUI, R., and KERMARREC, A.-M.: "The Many

Faces of Publish/Subscribe." *ACM Comput. Surv.*, (35)2:114-131, June 2003.

EUGSTER, P., GUERRAOUI, R., KERMARREC, A.-M., and MASSOULI'E, L.: "Epidemic Information Dissemination in Distributed Systems." *IEEE Computer*, (37)5:60-67, May 2004.

FARMER, W. M., GUTTMAN, J. D., and SWARUP, V.: "Security for Mobile Agents: Issues and Requirements." *Proc. 19th National Information Systems Security Conf.*, 1996. pp. 591-597.

FELBER, P. and NARASIMHAN, P.: "Experiences, Strategies, and Challenges in Building Fault-Tolerant CORBA Systems." *IEEE Computer*, (53)5:497-511, May 2004.

FERGUSON, N. and SCHNEIER, B.: *Practical Cryptography.* New York: John Wiley, 2003.

FIELDING, R., GETTYS, J., MOGUL, J., FRYSTYK, H., MASINTER, L., LEACH, P., and BERNERS-LEE, T.: "Hypertext Transfer Protocol – HTTP/1.1." RFC 2616, June 1999.

FIELDING, R. T. and TAYLOR, R. N.: "Principled Design of the Modern Web Architecture." *ACM Trans. Internet Techn.*, (2)2:115–150, 2002.

FILMAN, R. E., ELRAD, T., CLARKE, S., and AKSIT, M. (eds.): *Aspect-Oriented Software Development*. Reading, MA: Addison-Wesley, 2005.

FISCHER, M., LYNCH, N., and PATTERSON, M.: "Impossibility of Distributed Consensus with one Faulty Processor." *J. ACM*, (32)2:374-382, Apr. 1985.

FLEURY, M. and REVERBEL, F.: "The JBoss Extensible Server." *Proc. Middleware 2003*, vol. 2672 of *Lect. Notes Comp. Sc.*, (Rio de Janeiro, Brazil). Berlin: Springer-Verlag, 2003. pp. 344 - 373.

FLOYD, S., JACOBSON, V., MCCANNE, S., LIU, C.-G., and ZHANG, L.: "A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing." *IEEE/ACM Trans. Netw.*, (5)6:784-803, Dec. 1997.

FOSTER, I. and KESSELMAN, C.: *The Grid 2: Blueprint for a New Computing Infrastructure*. San Mateo, CA: Morgan Kaufman, 2nd ed., 2003.

FOSTER, I., KESSELMAN, C., TSUDIK, G., and TUECKE, S.: "A Security Architecture for Computational Grids." *Proc. Fifth Conf. Computer and Communications Security.* ACM, 1998. pp. 83-92.

FOSTER, I., KESSELMAN, C., and TUECKE, S.: "The Anatomy of the Grid, Enabling Scalable Virtual Organizations." *Journal of Supercomputer Applications*, (15)3:200-222, Fall 2001.

FOSTER, I., KISHIMOTO, H., and SAVVA, A.: "The Open Grid Services Architecture, Version 1.0." GGF Informational Document GFD-I.030, Jan. 2005.

FOWLER, R.: *Decentralized Object Finding Using Forwarding Addresses*. Ph.D. Thesis, University of Washington, Seattle, 1985.

FRANKLIN, M. J., CAREY, M. J., and LIVNY, M.: "Transactional Client-Server Cache Consistency: Alternatives and Performance." *ACM Trans. Database Syst.*, (22)3:315-363, Sept. 1997.

FREEMAN, E., HUPFER, S., and ARNOLD, K.: *JavaSpaces, Principles, Patterns and Practice*. Reading, MA: Addison-Wesley, 1999.

FREUND, R.: "Web Services Coordination, Version 1.0, Feb. 2005.

FRIEDMAN, R. and KAMA, A.: "Transparent Fault-Tolerant Java Virtual Machine." *Proc. 22nd Symp. on Reliable Distributed Systems*, (Florence, Italy). IEEE Computer Society Press: IEEE Computer Society Press, 2003. pp. 319-328.

FUGGETTA, A., PICCO, G. P., and VIGNA, G.: "Understanding Code Mobility." *IEEE Trans. Softw. Eng.*, (24)5:342-361, May 1998.

GAMMA, E., HELM, R., JOHNSON, R., and VLISSIDES, J.: *Design Patterns, Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1994.

GARBACKI, P., EPEMA, D., and VAN STEEN, M.: "A Two-Level Semantic Caching Scheme for Super-Peer Networks." *Proc. Tenth Web Caching Workshop*, (Sophia Antipolis, France). IEEE, 2005.

GARCIA-MOLINA, H.: "Elections in a Distributed Computing System." *IEEE Trans. Comp.*, (31)1:48-59, Jan. 1982.

GARMAN, J.: *Kerberos: The Definitive Guide*. Sebastopol, CA: O'Reilly & Associates, 2003.

GELERNTER, D.: "Generative Communication in Linda." *ACM Trans. Prog. Lang. Syst.*, (7)1:80-112, 1985.

GELERNTER, D. and CARRIERO, N.: "Coordination Languages and their Significance." *Commun. ACM*, (35)2:96-107, Feb. 1992.

GHEMAWAT, S., GOBIOFF, H., and LEUNG, S.-T.: "The Google File System." *Proc. 19th Symp. Operating System Principles*, (Bolton Landing, NY). New York, NY: ACM Press, 2003. pp. 29-43.

GIFFORD, D.: "Weighted Voting for Replicated Data." *Proc. Seventh Symp. Operating System Principles*. ACM, 1979. pp. 150-162.

GIGASPACES: *GigaSpaces Cache 5.0 Documentation*. New York, NY, 2005.

GIL, T. M. and POLETTO, M.: "MULTOPS: a Data-Structure for Bandwidth Attack Detection." *Proc. Tenth USENIX Security Symp.*, (Washington, DC). Berkeley, CA: USENIX, 2001. pp. 23-38.

GLADNEY, H.: "Access Control for Large Collections." *ACM Trans. Inf. Syst.*, (15)2:154-194, Apr. 1997.

GOLAND, Y., WHITEHEAD, E., FAIZI, A., CARTER, S., and JENSEN, D.: "HTTP Extensions for Distributed Authoring – WEBDAV." RFC 2518, Feb. 1999.

GOLLMANN, D.: *Computer Security*. New York: John Wiley, 2nd ed., 2006.

GONG, L. and SCHEMERS, R.: "Implementing Protection Domains in the Java Development Kit 1.2." *Proc. Symp. Network and Distributed System Security*. Internet Society, 1998. pp. 125-134.

GOPALAKRISHNAN, V., SILAGHI, B., BHATTACHARJEE, B., and KELEHER, P.: "Adaptive Replication in Peer-to-Peer Systems." *Proc. 24th Int'l Conf. on Distributed Computing Systems*, (Tokyo). Los Alamitos, CA: IEEE Computer Society Press, 2004. pp. 360-369.

GRAY, C. and CHERITON, D.: "Leases: An Efficient Fault-Tolerant Mechanism for Dis-

tributed File Cache Consistency." *Proc. 12th Symp. Operating System Principles*, (Litchfield Park, AZ). New York, NY: ACM Press, 1989. pp. 202-210.

GRAY, J., HELLAND, P., O'NEIL, P., and SASHNA, D.: "The Dangers of Replication and a Solution." *Proc. SIGMOD Int'l Conf. on Management Of Data*. ACM, 1996. pp. 173-182.

GRAY, J. and REUTER, A.: *Transaction Processing: Concepts and Techniques*. San Mateo, CA: Morgan Kaufman, 1993.

GRAY, J.: "Notes on Database Operating Systems." In Bayer, R., Graham, R., and Seegmuller, G. (eds.), *Operating Systems: An Advanced Course*, vol. 60 of *Lect. Notes Comp. Sc.*, pp. 393-481. Berlin: Springer-Verlag, 1978.

GRIMM, R., DAVIS, J., LEMAR, E., MACBETH, A., SWANSON, S., ANDERSON, T., BERSHAD, B., BORRIELLO, G., GRIBBLE, S., and WETHERALL, D.: "System Support for Pervasive Applications." *ACM Trans. Comp. Syst.*, (22)4:421-486, Nov. 2004.

GROPP, W., HUSS-LEDERMAN, S., LUMSDAINE, A., LUSK, E., NITZBERG, B., SAPHIR, W., and SNIR, M.: *MPI: The Complete Reference – The MPI-2 Extensions*. Cambridge, MA: MIT Press, 1998a.

GROPP, W., LUSK, E., and SKJELLUM, A.: *Using MPI, Portable Parallel Programming with the Message-Passing Interface*. Cambridge, MA: MIT Press, 2nd ed., 1998b.

GROSSKURTH, A. and GODFREY, M. W.: "A Reference Architecture for Web Browsers." *Proc. 21st Int'l Conf. Softw. Mainten.*, (Budapest, Hungary). Los Alamitos, CA: IEEE Computer Society Press, 2005. pp. 661-664.

GUDGIN, M., HADLEY, M., MENDELSOHN, N., MOREAU, J.-J., and NIELSEN, H. F.: "SOAP Version 1.2." W3C Recommendation, June 2003.

GUERRAOUI, R. and RODRIGUES, L.: *Introduction to Reliable Distributed Programming*. Berlin: Springer-Verlag, 2006.

GUERRAOUI, R. and SCHIPER, A.: "Software-Based Replication for Fault Tolerance." *IEEE Computer*, (30)4:68-74, Apr. 1997.

GUICHARD, J., FAUCHEUR, F. L., and VASSEUR, J.-P.: *Definitive MPLS Network Designs*. Indianapolis, IN: Cisco Press, 2005.

GULBRANDSEN, A., VIXIE, P., and ESIBOV, L.: "A dns rr for specifying the location of services (dns srv)." RFC 2782, Feb. 2000.

GUPTA, A., SAHIN, O. D., AGRAWAL, D., and ABBADI, A. E.: "Meghdoot: Content-Based Publish/Subscribe over P2P Networks." *Proc. Middleware 2004*, vol. 3231 of *Lect. Notes Comp. Sc.*, (Toronto, Canada). Berlin: Springer-Verlag, 2004. pp. 254-273.

GUSELLA, R. and ZATTI, S.: "The Accuracy of the Clock Synchronization Achieved by TEMPO in Berkeley UNIX 4.3BSD." *IEEE Trans. Softw. Eng.*, (15)7:847-853, July 1989.

HADZILACOS, V. and TOUEG, S.: "Fault-Tolerant Broadcasts and Related Problems." In Mullender, S. (ed.), *Distributed Systems*, pp. 97-145. Wokingham: Addison-Wesley, 2nd ed., 1993.

HALSALL, F.: *Multimedia Communications: Applications, Networks, Protocols and Standards*. Reading, MA: Addison-Wesley, 2001.

HANDURUKANDE, S., KERMARREC, A.-M., FESSANT, F. L., and MASSOULIE, L.: "Exploiting Semantic Clustering in the eDonkey P2P network." *Proc. 11th SIGOPS European Workshop*, (Leuven, Belgium). New York, NY: ACM Press, 2004.

HELDER, D. A. and JAMIN, S.: "End-Host Multicast Communication Using Switch-Trees Protocols." *Proc. Second Int'l Symp. Cluster Comput. & Grid*, (Berlin, Germany). Los Alamitos, CA: IEEE Computer Society Press, 2002. pp. 419-424.

HELLERSTEIN, J. L., DIAO, Y., PAREKH, S., and TILBURY, D. M.: *Feedback Control of Computing Systems*. New York: John Wiley, 2004.

HENNING, M.: "A New Approach to Object-Oriented Middleware." *IEEE Internet Comput.*, (8)1:66-75, Jan. 2004.

HENNING, M.: "The Rise and Fall of CORBA." *ACM Queue*, (4)5, 2006.

HENNING, M. and SPRUIELL, M.: *Distributed Programming with Ice*. ZeroC Inc., Brisbane, Australia, May 2005.

HENNING, M. and VINOSKI, S.: *Advanced CORBA Programming with C++*. Reading, MA: Addison-Wesley, 1999.

HOCHSTETLER, S. and BERINGER, B.: "Linux Clustering with CSM and GPFS." Technical Report SG24-6601-02, International Technical Support Organization, IBM, Austin, TX, Jan. 2004.

HOHPE, G. and WOOLF, B.: *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Reading, MA: Addison-Wesley, 2004.

HOROWITZ, M. and LUNT, S.: "FTP Security Extensions." RFC 2228, Oct. 1997.

HOWES, T.: "The String Representation of LDAP Search Filters." RFC 2254, Dec. 1997.

HUA CHU, Y., RAO, S. G., SESHAN, S., and ZHANG, H.: "A Case for End System Multicast." *IEEE J. Selected Areas Commun.*, (20)8:1456-1471, Oct. 2002.

HUFFAKER, B., FOMENKOV, M., PLUMMER, D. J., MOORE, D., and CLAFFY, K.: "Distance Metrics in the Internet." *Proc. Int'l Telecommun. Symp.*, (Natal RN, Brazil). Los Alamitos, CA: IEEE Computer Society Press, 2002.

HUNT, G., NAHUM, E., and TRACEY, J.: "Enabling Content-Based Load Distribution for Scalable Services." Technical Report, IBM T.J. Watson Research Center, May 1997.

HUTTO, P. and AHAMAD, M.: "Slow Memory: Weakening Consistency to Enhance Concurrency in Distributed Shared Memories." *Proc. Tenth Int'l Conf. on Distributed Computing Systems*. IEEE, 1990. pp. 302-311.

IBM: *WebSphere MQ Application Programming Guide*, May 2005a.

IBM: *WebSphere MQ Intercommunication*, May 2005b.

IBM: *WebSphere MQ Publish/Subscribe User's Guide*, May 2005c.

IBM: *WebSphere MQ System Administration*, May 2005d.

ISO: "Open Distributed Processing Reference Model." International Standard ISO/IEC IS 10746, 1995.

JAEGER, T., PRAKASH, A., LIEDTKE, J., and ISLAM, N.: "Flexible Control of Downloaded Executable Content." *ACM Trans. Inf. Syst. Sec.*, (2)2:177-228, May 1999.

JALOTE, P.: *Fault Tolerance in Distributed Systems*. Englewood Cliffs, NJ: Prentice Hall, 1994.

JANIC, M.: *Multicast in Network and Application Layer*. Ph.d. Thesis, Delft University of Technology, The Netherlands, Oct. 2005.

JANIGA, M. J., DIBNER, G., and GOVERNALI, F. J.: "Internet Infrastructure: Content Delivery." Goldman Sachs Global Equity Research, Apr. 2001.

JELASITY, M., GUERRAOUI, R., KERMARREC, A.-M., and VAN STEEN, M.: "The Peer Sampling Service: Experimental Evaluation of Unstructured Gossip-Based Implementations." *Proc. Middleware 2004*, vol. 3231 of *Lect. Notes Comp. Sc.*, (Toronto, Canada). Berlin: Springer-Verlag, 2004. pp. 79-98.

JELASITY, M., VOULGARIS, S., GUERRAOUI, R., KERMARREC, A.-M., and VAN STEEN, M.: "Gossip-based Peer Sampling." Technical Report, Vrije Universiteit, Department of Computer Science, Sept. 2005a.

JELASITY, M. and BABAOGLU, O.: "T-Man: Gossip-based Overlay Topology Management." *Proc. Third Int'l Workshop Eng. Self-Organising App.*, (Utrecht, The Netherlands), 2005.

JELASITY, M., MONTRESOR, A., and BABAOGLU, O.: "Gossip-based Aggregation in Large Dynamic Networks." *ACM Trans. Comp. Syst.*, (23)3:219-252, Aug. 2005b.

JIN, J. and NAHRSTEDT, K.: "QoS Specification Languages for Distributed Multimedia Applications: A Survey and Taxonomy." *IEEE Multimedia*, (11)3:74-87, July 2004.

JING, J., HELAL, A., and ELMAGARMID, A.: "Client-Server Computing in Mobile Environments." *ACM Comput. Surv.*, (31)2:117-157, June 1999.

JOHNSON, B.: "An Introduction to the Design and Analysis of Fault-Tolerant Systems." In Pradhan, D.K. (ed.), *Fault-Tolerant Computer System Design*, pp. 1-87. Upper Saddle River, NJ: Prentice Hall, 1995.

JOHNSON, D., PERKINS, C., and ARKKO, J.: "Mobility Support for IPv6." RFC 3775, June 2004.

JOSEPH, J., ERNEST, M., and FELLENSTEIN, C.: "Evolution of grid computing architecture and grid adoption models." *IBM Syst. J.*, (43)4:624-645, Apr. 2004.

JUL, E., LEVY, H., HUTCHINSON, N., and BLACK, A.: "Fine-Grained Mobility in the Emerald System." *ACM Trans. Comp. Syst.*, (6)1:109-133, Feb. 1988.

JUNG, J., SIT, E., BALAKRISHNAN, H., and MORRIS, R.: "DNS Performance and the Effectiveness of Caching." *IEEE/ACM Trans. Netw.*, (10)5:589 - 603, Oct. 2002.

KAHN, D.: *The Codebreakers*. New York: Macmillan, 1967.

KAMINSKY, M., SAVVIDES, G., MAZIhRES, D., and KAASHOEK, M. F.: "Decentralized User Authentication in a Global File System." *Proc. 19th Symp. Operating System Principles*, (Bolton Landing, NY). New York, NY: ACM Press, 2003. pp. 60-73.

KANTARCIOGLU, M. and CLIFTON, C.: "Security Issues in Querying Encrypted Data." *Proc. 19th Conf. Data & Appl. Security*, vol. 3654 of *Lect. Notes Comp. Sc.*, (Storrs, CT). Berlin: Springer-Verlag, 2005. pp. 325-337.

KARNIK, N. and TRIPATHI, A.: "Security in the Ajanta Mobile Agent System." *Software – Practice & Experience*, (31)4:301-329, Apr. 2001.

KASERA, S., KUROSE, J., and TOWSLEY, D.: "Scalable Reliable Multicast Using Multi-

ple Multicast Groups." *Proc. Int'l Conf. Measurements and Modeling of Computer Systems.* ACM, 1997. pp. 64-74.

KATZ, E., BUTLER, M., and MCGRATH, R.: "A Scalable HTTP Server: The NCSA Prototype." *Comp. Netw. & ISDN Syst.*, (27)2:155-164, Sept. 1994.

KAUFMAN, C., PERLMAN, R., and SPECINER, M.: *Network Security: Private Communication in a Public World.* Englewood Cliffs, NJ: Prentice Hall, 2nd ed., 2003.

KENT, S.: "Internet Privacy Enhanced Mail." *Commun. ACM*, (36)8:48-60, Aug. 1993.

KEPHART, J. O. and CHESS, D. M.: "The Vision of Autonomic Computing." *IEEE Computer*, (36)1:41-50, Jan. 2003.

KHOSHAFIAN, S. and BUCKIEWICZ, M.: *Introduction to Groupware, Workflow, and Workgroup Computing.* New York: John Wiley, 1995.

KHURANA, H. and KOLEVA, R.: "Scalable Security and Accounting Services for Content-Based Publish Subscribe Systems." *Int'l J. E-Business Res.*, (2), 2006.

KIM, S., PAN, K., SINDERSON, E., and WHITEHEAD, J.: "Architecture and Data Model of a WebDAV-based Collaborative System." *Proc. Collaborative Techn. Symp.\fR, (San Diego, CA), 2004. pp. 48-55.*

KISTLER, J. and SATYANARYANAN, M.: "Disconnected Operation in the Coda File System." *ACM Trans. Comp. Syst.*, (10)1:3-25, Feb. 1992.

KLEIMAN, S.: "Vnodes: an Architecture for Multiple File System Types in UNIX." *Proc. Summer Techn. Conf.* USENIX, 1986. pp. 238-247.

KOHL, J., NEUMAN, B., and T'SO, T.: "The Evolution of the Kerberos Authentication System." In Brazier, F. and Johansen, D. (eds.), *Distributed Open Systems*, pp. 78-94. Los Alamitos, CA: IEEE Computer Society Press, 1994.

KON, F., COSTA, F., CAMPBELL, R., and BLAIR, G.: "The Case for Reflective Middleware." *Commun. ACM*, (45)6:33-38, June 2002.

KOPETZ, H. and VERISSIMO, P.: "Real Time and Dependability Concepts." In Mullender, S. (ed.), *Distributed Systems*, pp. 411-446. Wokingham: Addison-Wesley, 2nd ed., 1993.

KOSTOULAS, M. G., MATSA, M., MENDELSOHN, N., PERKINS, E., HEIFETS, A., and MERCALDI, M.: "XML Screamer: An Integrated Approach to High Performance XML Parsing, Validation and Deserialization." *Proc. 15th Int'l WWW Conf.*, (Edinburgh, Scotland). New York, NY: ACM Press, 2006.

KUMAR, P. and SATYANARAYANAN, M.: "Flexible and Safe Resolution of File Conflicts." *Proc. Winter Techn. Conf.* USENIX, 1995. pp. 95-106.

LAI, A. and NIEH, J.: "Limits of Wide-Area Thin-Client Computing." *Proc. Int'l Conf. Measurements and Modeling of Computer Systems*, (Marina Del Rey, CA). New York, NY: ACM Press, 2002. pp. 228-239.

LAMACCHIA, B. and ODLYZKO, A.: "Computation of Discrete Logarithms in Prime Fields." *Designs, Codes, and Cryptography*, (1)1:47-62, May 1991.

LAMPORT, L.: "Time, Clocks, and the Ordering of Events in a Distributed System." *Commun. ACM*, (21)7:558-565, July 1978.

LAMPORT, L.: "How to Make a Multiprocessor Computer that Correctly Executes Multiprocessor Programs." *IEEE Trans. Comp.*, (C-29)9:690-691, Sept. 1979.

LAMPORT, L., SHOSTAK, R., and PAESE, M.: "Byzantine Generals Problem." *ACM Trans. Prog. Lang. Syst.*, (4)3:382-401, July 1982.

LAMPSON, B., ABADI, M., BURROWS, M., and WOBBER, E.: "Authentication in Distributed Systems: Theory and Practice." *ACM Trans. Comp. Syst.*, (10)4:265-310, Nov. 1992.

LAPRIE, J.-C.: "Dependability – Its Attributes, Impairments and Means." In Randell, B., Laprie, J.-C., Kopetz, H., and Littlewood, B. (eds.), *Predictably Dependable Computing Systems*, pp. 3-24. Berlin: Springer-Verlag, 1995.

LAURIE, B. and LAURIE, P.: *Apache: The Definitive Guide.* Sebastopol, CA: O'Reilly & Associates, 3rd ed., 2002.

LEFF, A. and RAYFIELD, J. T.: "Alternative Edge-server Architectures for Enterprise JavaBeans Applications." *Proc. Middleware 2004*, vol. 3231 of *Lect. Notes Comp. Sc.*, (Toronto, Canada). Berlin: Springer-Verlag, 2004. pp. 195-211.

LEIGHTON, F. and LEWIN, D.: "Global Hosting System." United States Patent, Number 6,108,703, Aug. 2000.

LEVIEN, R. (ed.): *Signposts in Cyberspace: The Domain Name System and Internet Navigation.* Washington, DC: National Academic Research Council, 2005.

LEVINE, B. and GARCIA-LUNA-ACEVES, J.: "A Comparison of Reliable Multicast Protocols." *ACM Multimedia Systems Journal*, (6)5:334-348, 1998.

LEWIS, B. and BERG, D. J.: *Multithreaded Programming with Pthreads.* Englewood Cliffs, NJ: Prentice Hall, 2nd ed., 1998.

LI, G. and JACOBSEN, H.-A.: "Composite Subscriptions in Content-Based Publish/Subscribe Systems." *Proc. Middleware 2005*, vol. 3790 of *Lect. Notes Comp. Sc.*, (Grenoble, France). Berlin: Springer-Verlag, 2005. pp. 249-269.

LI, J., LU, C., and SHI, W.: "An Efficient Scheme for Preserving Confidentiality in Content-Based Publish-Subscribe Systems." Technical Report GIT-CC-04-01, Georgia Institute of Technology, College of Computing, 2004a.

LI, N., MITCHELL, J. C., and TONG, D.: "Securing Java RMI-based Distributed Applications." *Proc. 20th Ann. Computer Security Application Conf.*, (Tucson, AZ). ACSA, 2004b.

LILJA, D.: "Cache Coherence in Large-Scale Shared-Memory Multiprocessors: Issues and Comparisons." *ACM Comput. Surv.*, (25)3:303-338, Sept. 1993.

LIN, M.-J. and MARZULLO, K.: "Directional Gossip: Gossip in a Wide-Area Network." In *Proc. Third European Dependable Computing Conf.*, vol. 1667 of *Lect. Notes Comp. Sc.*, pp. 364-379. Berlin: Springer-Verlag, Sept. 1999.

LIN, S.-D., LIAN, Q., CHEN, M., , and ZHANG, Z.: "A Practical Distributed Mutual Exclusion Protocol in Dynamic Peer-to-Peer Systems." *Proc. Third Int'l Workshop on Peer-to-Peer Systems*, vol. 3279 of *Lect. Notes Comp. Sc.*, (La Jolla, CA). Berlin: Springer-Verlag, 2004. pp. 11-21.

LING, B. C., KICIMAN, E., and FOX, A.: "Session State: Beyond Soft State." *Proc. First Symp. Networked Systems Design and Impl.*, (San Francisco, CA). Berkeley, CA:

USENIX, 2004. pp. 295-308.

LINN, J.: "Generic Security Service Application Program Interface, version 2." RFC 2078, Jan. 1997.

LIU, C.-G., ESTRIN, D., SHENKER, S., and ZHANG, L.: "Local Error Recovery in SRM: Comparison of Two Approaches." *IEEE/ACM Trans. Netw.*, (6)6:686-699, Dec. 1998.

LIU, H. and JACOBSEN, H.-A.: "Modeling Uncertainties in Publish/Subscribe Systems." *Proc. 20th Int'l Conf. Data Engineering*, (Boston, MA). Los Alamitos, CA: IEEE Computer Society Press, 2004. pp. 510-522.

LO, V., ZHOU, D., LIU, Y., GauthierDickey, C., and LI, J.: "Scalable Supernode Selection in Peer-to-Peer Overlay Networks." *Proc. Second Hot Topics in Peer-to-Peer Systems*, (La Jolla, CA), 2005.

LOSHIN, P. (ed.): *Big Book of Lightweight Directory Access Protocol (LDAP) RFCs.* San Mateo, CA: Morgan Kaufman, 2000.

LUA, E. K., CROWCROFT, J., PIAS, M., SHARMA, R., and LIM, S.: "A Survey and Comparison of Peer-to-Peer Overlay Network Schemes." *IEEE Communications Surveys & Tutorials*, (7)2:22-73, Apr. 2005.

LUI, J., MISRA, V., and RUBENSTEIN, D.: "On the Robustness of Soft State Protocols." *Proc. 12th Int'l Conf. on Network Protocols*, (Berlin, Germany). Los Alamitos, CA: IEEE Computer Society Press, 2004. pp. 50-60.

LUOTONEN, A. and ALTIS, K.: "World-Wide Web Proxies." *Comp. Netw. & ISDN Syst.*, (27)2:1845-1855, 1994.

LYNCH, N.: *Distributed Algorithms.* San Mateo, CA: Morgan Kaufman, 1996.

MAASSEN, J., KIELMANN, T., and BAL, H. E.: "Parallel Application Experience with Replicated Method Invocation." *Conc. & Comput.: Prac. Exp.*, (13)8-9:681-712, 2001.

MACGREGOR, R., DURBIN, D., OWLETT, J., and YEOMANS, A.: *Java Network Security.* Upper Saddle River, NJ: Prentice Hall, 1998.

MADDEN, S. R., FRANKLIN, M. J., HELLERSTEIN, J. M., and HONG, W.: "TinyDB: An Acquisitional Query Processing System for Sensor Networks." *ACM Trans. Database Syst.*, (30)1:122-173, 2005.

MAKPANGOU, M., GOURHANT, Y., LE NARZUL, J.-P., and SHAPIRO, M.: "Fragmented Objects for Distributed Abstractions." In Casavant, T. and Singhal, M. (eds.), *Readings in Distributed Computing Systems*, pp. 170-186. Los Alamitos, CA: IEEE Computer Society Press, 1994.

MALKHI, D. and REITER, M.: "Secure Execution of Java Applets using a Remote Playground." *IEEE Trans. Softw. Eng.*, (26)12:1197-1209, Dec. 2000.

MAMEI, M. and ZAMBONELLI, F.: "Programming Pervasive and Mobile Computing Applications with the TOTA Middleware." *Proc. Second Int'l Conf. Pervasive Computing and Communications (PerCom)*, (Orlando, FL). Los Alamitos, CA: IEEE Computer Society Press, 2004. pp. 263-273.

MANOLA, F. and MILLER, E.: "RDF Primer." W3C Recommendation, Feb. 2004.

MARCUS, E. and STERN, H.: *Blueprints for High Availability.* New York: John Wiley, 2nd ed., 2003.

MASCOLO, C., CAPRA, L., and EMMERICH, W.: "Principles of Mobile Computing Middleware." In Mahmoud, Qusay H. (ed.), *Middleware for Communications*, chapter 12. New York: John Wiley, 2004.

MASINTER, L.: "The Data URL Scheme." RFC 2397, Aug. 1998.

MAZIERES, D., KAMINSKY, M., KAASHOEK, M., and WITCHEL, E.: "Separating Key Management from File System Security." *Proc. 17th Symp. Operating System Principles.* ACM, 1999. pp. 124-139.

MAZOUNI, K., GARBINATO, B., and GUERRAOUI, R.: "Building Reliable Client-Server Software Using Actively Replicated Objects." In Graham, I., Magnusson, B., Meyer, B., and Nerson, J.-M (eds.), *Technology of Object Oriented Languages and Systems*, pp. 37-53. Englewood Cliffs, NJ: Prentice Hall, 1995.

MCKINLEY, P., SADJADI, S., KASTEN, E., and CHENG, B.: "Composing Adaptive Software." *IEEE Computer*, (37)7:56-64, Jan. 2004.

MEHTA, N., MEDVIDOVIC, N., and PHADKE, S.: "Towards A Taxonomy Of Software Connectors." *Proc. 22nd Int'l Conf. on Software Engineering*, (Limerick, Ireland). New York, NY: ACM Press, 2000. pp. 178-187.

MENEZES, A. J., VAN OORSCHOT, P. C., and VANSTONE, S. A.: *Handbook of Applied Cryptography.* Boca Raton: CRC Press, 3rd ed., 1996.

MERIDETH, M. G., IYENGAR, A., MIKALSEN, T., TAI, S., ROUVELLOU, I., and NARASIMHAN, P.: "Thema: Byzantine-Fault-Tolerant Middleware for Web-Service Applications." *Proc. 24th Symp. on Reliable Distributed Systems*, (Orlando, FL). Los Alamitos, CA: IEEE Computer Society Press, 2005. pp. 131-142.

MEYER, B.: *Object-Oriented Software Construction.* Englewood Cliffs, NJ: Prentice Hall, 2nd ed., 1997.

MILLER, B. N., KONSTAN, J. A., and RIEDL, J.: "PocketLens: Toward a Personal Recommender System." *ACM Trans. Inf. Syst.*, (22)3:437–476, July 2004.

MILLS, D. L.: *Computer Network Time Synchronization: The Network Time Protocol.* Boca Raton, FL: CRC Press, 2006.

MILLS, D. L.: "Network Time Protocol (version 3): Specification, Implementation, and Analysis." RFC 1305, July 1992.

MILOJICIC, D., DOUGLIS, F., PAINDAVEINE, Y., WHEELER, R., and ZHOU, S.: "Process Migration." *ACM Comput. Surv.*, (32)3:241-299, Sept. 2000.

MIN, S. L. and BAER, J.-L.: "Design and Analysis of a Scalable Cache Coherence Scheme Based on Clocks and Timestamps." *IEEE Trans. Par. Distr. Syst.*, (3)1:25-44, Jan. 1992.

MIRKOVIC, J., DIETRICH, S., and ANDPETER REIHER, D. D.: *Internet Denial of Service: Attack and Defense Mechanisms.* Englewood Cliffs, NJ: Prentice Hall, 2005.

MIRKOVIC, J. and REIHER, P.: "A Taxonomy of DDoS Attack and DDoS Defense Mechanisms." *ACM Comp. Commun. Rev.*, (34)2:39–53, Apr. 2004.

MOCKAPETRIS, P.: "Domain Names - Concepts and Facilities." RFC 1034, Nov. 1987.

MONSON-HAEFEL, R., BURKE, B., and LABOUREY, S.: *Enterprise Java Beans.* Sebastopol, CA: O'Reilly & Associates, 4th ed., 2004.

MOSER, L., MELLIAR-SMITH, P., AGARWAL, D., BUDHIA, R., and LINGLEY-

PAPADOPOULOS, C.: "Totem: A Fault-Tolerant Multicast Group Communication System." *Commun. ACM*, (39)4:54-63, Apr. 1996.

MOSER, L., MELLIOR-SMITH, P., and NARASIMHAN, P.: "Consistent Object Replication in the Eternal System." *Theory and Practice of Object Systems*, (4)2:81-92, 1998.

MULLENDER, S. and TANENBAUM, A.: "Immediate Files." *Software – Practice & Experience*, (14)3:365-368, 1984.

MUNTZ, D. and HONEYMAN, P.: "Multi-level Caching in Distributed File Systems." *Proc. Winter Techn. Conf.* USENIX, 1992. pp. 305-313.

MURPHY, A., PICCO, G., and ROMAN, G.-C.: "Lime: A Middleware for Physical and Logical Mobility." *Proc. 21st Int'l Conf. on Distr. Computing Systems*, (Phoenix, AZ). Los Alamitos, CA: IEEE Computer Society Press, 2001. pp. 524-533.

MUTHITACHAROEN, A., MORRIS, R., GIL, T., and CHEN, B.: "Ivy: A Read/Write Peer-to-Peer File System." *Proc. Fifth Symp. on Operating System Design and Implementation*, (Boston, MA). New York, NY: ACM Press, 2002. pp. 31-44.

NAPPER, J., ALVISI, L., and VIN, H. M.: "A Fault-Tolerant Java Virtual Machine." *Proc. Int'l Conf. Dependable Systems and Networks*, (San Francisco, CA). Los Alamitos, CA: IEEE Computer Society Press, 2003. pp. 425-434.

NARASIMHAN, P., MOSER, L., and MELLIAR-SMITH, P.: "The Eternal System." In Urban, J. and Dasgupta, P. (eds.), *Encyclopedia of Distributed Computing*. Dordrecht, The Netherlands: Kluwer Academic Publishers, 2000.

NAYATE, A., DAHLIN, M., and IYENGAR, A.: "Transparent Information Dissemination." *Proc. Middleware 2004*, vol. 3231 of *Lect. Notes Comp. Sc.*, (Toronto, Canada). Berlin: Springer-Verlag, 2004. pp. 212-231.

NEEDHAM, R. and SCHROEDER, M.: "Using Encryption for Authentication in Large Networks of Computers." *Commun. ACM*, (21)12:993-999, Dec. 1978.

NEEDHAM, R.: "Names." In Mullender, S. (ed.), *Distributed Systems*, pp. 315-327. Wokingham: Addison-Wesley, 2nd ed., 1993.

NELSON, B.: *Remote Procedure Call.* Ph.D. Thesis, Carnegie-Mellon University, 1981.

NEUMAN, B.: "Scale in Distributed Systems." In Casavant, T. and Singhal, M. (eds.), *Readings in Distributed Computing Systems*, pp. 463-489. Los Alamitos, CA: IEEE Computer Society Press, 1994.

NEUMAN, B.: "Proxy-Based Authorization and Accounting for Distributed Systems." *Proc. 13th Int'l Conf. on Distributed Computing Systems*. IEEE, 1993. pp. 283-291.

NEUMAN, C., YU, T., HARTMAN, S., and RAEBURN, K.: "The Kerberos Network Authentication Service." RFC 4120, July 2005.

NEUMANN, P.: "Architectures and Formal Representations for Secure Systems." Technical Report, Computer Science Laboratory, SRI International, Menlo Park, CA, Oct. 1995.

NG, E. and ZHANG, H.: "Predicting Internet Network Distance with Coordinates-Based Approaches." *Proc. 21st INFOCOM Conf.*, (New York, NY). Los Alamitos, CA: IEEE Computer Society Press, 2002.

NIEMELA, E. and LATVAKOSKI, J.: "Survey of Requirements and Solutions for Ubiqui-

tous Software." *Proc. Third Int'l Conf. Mobile & Ubiq. Multimedia*, (College Park, MY), 2004. pp. 71–78.

NOBLE, B., FLEIS, B., and KIM, M.: "A Case for Fluid Replication." *Proc. NetStore'99*, 1999.

OBRACZKA, K.: "Multicast Transport Protocols: A Survey and Taxonomy." *IEEE Commun. Mag.*, (36)1:94-102, Jan. 1998.

OMG: "The Common Object Request Broker: Core Specification, revision 3.0.3." OMG Document formal/04-03-12, Object Management Group, Framingham, MA, Mar. 2004a.

OMG: "UML 2.0 Superstructure Specification." OMG Document ptc/04-10-02, Object Management Group, Framingham, MA, Oct. 2004b.

OPPENHEIMER, D., ALBRECHT, J., PATTERSON, D., and VAHDAT, A.: "Design and Implementation Tradeoffs for Wide-Area Resource Discovery." *Proc. 14th Int'l Symp. on High Performance Distributed Computing*, (Research Triangle Park, NC). Los Alamitos, CA: IEEE Computer Society Press, 2005.

ORAM, A. (ed.): *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*. Sebastopol, CA: O'Reilly & Associates, 2001.

OZSU, T. and VALDURIEZ, P.: *Principles of Distributed Database Systems*. Upper Saddle River, NJ: Prentice Hall, 2nd ed., 1999.

PAI, V., ARON, M., BANGA, G., SVENDSEN, M., DRUSCHEL, P., ZWAENEPOEL, W., and NAHUM, E.: "Locality-Aware Request Distribution in Cluster-Based Network Servers." *Proc. Eighth Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, (San Jose, CA). New York, NY: ACM Press, 1998. pp. 205-216.

PANZIERI, F. and SHRIVASTAVA, S.: "Rajdoot: A Remote Procedure Call Mechanism with Orphan Detection and Killing." *IEEE Trans. Softw. Eng.*, (14)1:30-37, Jan. 1988.

PARTRIDGE, C., MENDEZ, T., and MILLIKEN, W.: "Host Anycasting Service." RFC 1546, Nov. 1993.

PATE, S.: *UNIX Filesystems: Evolution, Design, and Implementation*. New York: John Wiley, 2003.

PEASE, M., SHOSTAK, R., and LAMPORT, L.: "Reaching Agreement in the Presence of Faults." *J. ACM*, (27)2:228-234, Apr. 1980.

PERKINS, C., HODSON, O., and HARDMAN, V.: "A Survey of Packet Loss Recovery Techniques for Streaming Audio." *IEEE Network*, (12)5:40-48, Sept. 1998.

PETERSON, L. and DAVIE, B.: *Computer Networks, A Systems Approach*. San Mateo, CA: Morgan Kaufman, 3rd ed., 2003.

PETERSON, L., BAVIER, A., FIUCZYNSKI, M., MUIR, S., and ROSCOE, T.: "Towards a Comprehensive PlanetLab Architecture." Technical Report PDN-05-030, PlanetLab Consortium, June 2005.

PFLEEGER, C.: *Security in Computing*. Upper Saddle River, NJ: Prentice Hall, 3rd ed., 2003.

PICCO, G., BALZAROTTI, D., and COSTA, P.: "LighTS: A Lightweight, Customizable Tuple Space Supporting Context-Aware Applications." *Proc. Symp. Applied Computing*, (Santa Fe, NM). New York, NY: ACM Press, 2005. pp. 413-419.

PIERRE, G. and VAN STEEN, M.: "Globule: A Collaborative Content Delivery Network." *IEEE Commun. Mag.*, (44)8, Aug. 2006.

PIERRE, G., VAN STEEN, M., and TANENBAUM, A.: "Dynamically Selecting Optimal Distribution Strategies for Web Documents." *IEEE Trans. Comp.*, (51)6:637-651, June 2002.

PIETZUCH, P. R. and BACON, J. M.: "Hermes: A Distributed Event-Based Middleware Architecture." *Proc. Workshop on Distributed Event-Based Systems*, (Vienna, Austria). Los Alamitos, CA: IEEE Computer Society Press, 2002.

PIKE, R., PRESOTTO, D., DORWARD, S., FLANDRENA, B., THOMPSON, K., TRICKEY, H., and WINTERBOTTOM, P.: "Plan 9 from Bell Labs." *Computing Systems*, (8)3:221-254, Summer 1995.

PINZARI, G.: "NX X Protocol Compression." Technical Report D-309/3-NXP-DOC, NoMachine, Rome, Italy, Sept. 2003.

PITOURA, E. and SAMARAS, G.: "Locating Objects in Mobile Computing." *IEEE Trans. Know. Data Eng.*, (13)4:571-592, July 2001.

PLAINFOSSE, D. and SHAPIRO, M.: "A Survey of Distributed Garbage Collection Techniques." In *Proc. Int'l Workshop on Memory Management*, vol. 986 of *Lect. Notes Comp. Sc.*, pp. 211-249. Berlin: Springer-Verlag, Sept. 1995.

PLUMMER, D.: "Ethernet Address Resolution Protocol." RFC 826, Nov. 1982.

PODLING, S. and BOSZORMENYI, L.: "A Survey of Web Cache Replacement Strategies." *ACM Comput. Surv.*, (35)4:374-398, Dec. 2003.

POPESCU, B., VAN STEEN, M., and TANENBAUM, A.: "A Security Architecture for Object-Based Distributed Systems." *Proc. 18th Ann. Computer Security Application Conf.*, (Las Vegas, NA). ACSA, 2002.

POSTEL, J.: "Simple Mail Transfer Protocol." RFC 821, Aug. 1982.

POSTEL, J. and REYNOLDS, J.: "File Transfer Protocol." RFC 995, Oct. 1985.

POTZL, H., ANDERSON, M., and STEINBRINK, B.: "Linux-VServer: Resource Efficient Context Isolation." *Free Software Magazine*, no. 5, June 2005.

POUWELSE, J., GARBACKI, P., EPEMA, D., and SIPS, H.: "A Measurement Study of the BitTorrent Peer-to-Peer File-Sharing System." Technical Report PDS-2004-003, Technical University Delft, Apr. 2004.

POUWELSE, J. A., GARBACKI, P., EPEMA, D. H. J., and SIPS, H. J.: "The Bittorrent P2P File-Sharing System: Measurements and Analysis." *Proc. Fourth Int'l Workshop on Peer-to-Peer Systems*, vol. 3640 of *Lect. Notes Comp. Sc.*, (Ithaca, NY). Berlin: Springer-Verlag, 2005. pp. 205-216.

QIN, F., TUCEK, J., SUNDARESAN, J., and ZHOU, Y.: "Rx: Treating Bugs as Allergies - A Safe Method to Survive Software Failures." *Proc. 20th Symp. Operating System Principles*, (Brighton, UK). New York, NY: ACM Press, 2005. pp. 235-248.

QIU, L., PADMANABHAN, V., and VOELKER, G.: "On the Placement of Web Server Replicas." *Proc. 20th INFOCOM Conf.*, (Anchorage (AK)). Los Alamitos, CA: IEEE Computer Society Press, 2001. pp. 1587-1596.

RABINOVICH, M. and SPASTSCHECK, O.: *Web Caching and Replication*. Reading, MA: Addison-Wesley, 2002.

RABINOVICH, M., RABINOVICH, I., RAJARAMAN, R., and AGGARWAL, A.: "A Dynamic Object Replication and Migration Protocol for an Internet Hosting Service." *Proc. 19th Int'l Conf. on Distributed Computing Systems*. IEEE, 1999. pp. 101-113.

RADIA, S.: *Names, Contexts, and Closure Mechanisms in Distributed Computing Environments*. Ph.D. Thesis, University of Waterloo, Ontario, 1989.

RADOSLAVOV, P., GOVINDAN, R., and ESTRIN, D.: "Topology-Informed Internet Replica Placement." *Proc. Sixth Web Caching Workshop*, (Boston, MA). Amsterdam: North-Holland, 2001.

RAFAELI, S. and HUTCHISON, D.: "A Survey of Key Management for Secure Group Communication." *ACM Comput. Surv.*, (35)3:309-329, Sept. 2003.

RAICIU, C. and ROSENBLUM, D.: "Enabling Confidentiality in Content-Based Publish/Subscribe Infrastructures." Technical Report RN/05/30, Department of Computer Science, University College London, 2005.

RAMANATHAN, P., SHIN, K., and BUTLER, R.: "Fault-Tolerant Clock Synchronization in Distributed Systems." *IEEE Computer*, (23)10:33-42, Oct. 1990.

RAMASUBRAMANIAN, V. and SIRER, E. G.: "The Design and Implementation of a Next Generation Name Service for the Internet." *Proc. SIGCOMM*, (Portland, OR). New York, NY: ACM Press, 2004a.

RAMASUBRAMANIAN, V. and SIRER, E. G.: "Beehive: O(1) Lookup Performance for Power-Law Query Distributions in Peer-to-Peer Overlays." *Proc. First Symp. Networked Systems Design and Impl.*, (San Francisco, CA). Berkeley, CA: USENIX, 2004b. pp. 99-112.

RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., and SCHENKER, S.: "A Scalable Content-Addressable Network." *Proc. SIGCOMM*. ACM, 2001. pp. 161-172.

RAYNAL, M. and SINGHAL, M.: "Logical Time: Capturing Causality in Distributed Systems." *IEEE Computer*, (29)2:49-56, Feb. 1996.

REITER, M.: "How to Securely Replicate Services." *ACM Trans. Prog. Lang. Syst.*, (16)3:986-1009, May 1994.

REITER, M., BIRMAN, K., and VAN RENESSE, R.: "A Security Architecture for Fault-Tolerant Systems." *ACM Trans. Comp. Syst.*, (12)4:340-371, Nov. 1994.

RESCORLA, E. and SCHIFFMAN, A.: "The Secure HyperText Transfer Protocol." RFC 2660, Aug. 1999.

REYNOLDS, J. and POSTEL, J.: "Assigned Numbers." RFC 1700, Oct. 1994.

RICART, G. and AGRAWALA, A.: "An Optimal Algorithm for Mutual Exclusion in Computer Networks." *Commun. ACM*, (24)1:9-17, Jan. 1981.

RISSON, J. and MOORS, T.: "Survey of Research towards Robust Peer-to-Peer Networks: Search Methods." *Comp. Netw.*, (50), 2006.

RIVEST, R.: "The MD5 Message Digest Algorithm." RFC 1321, Apr. 1992.

RIVEST, R., SHAMIR, A., and ADLEMAN, L.: "A Method for Obtaining Digital Signatures and Public-key Cryptosystems." *Commun. ACM*, (21)2:120-126, Feb. 1978.

RIZZO, L.: "Effective Erasure Codes for Reliable Computer Communication Protocols." *ACM Comp. Commun. Rev.*, (27)2:24-36, Apr. 1997.

RODRIGUES, L., FONSECA, H., and VERISSIMO, P.: "Totally Ordered Multicast in Large-Scale Systems." *Proc. 16th Int'l Conf. on Distributed Computing Systems*. IEEE, 1996. pp. 503-510.

RODRIGUES, R. and LISKOV, B.: "High Availability in DHTs: Erasure Coding vs. Replication." *Proc. Fourth Int'l Workshop on Peer-to-Peer Systems*, (Ithaca, NY), 2005.

RODRIGUEZ, P., SPANNER, C., and BIERSACK, E.: "Analysis of Web Caching Architecture: Hierarchical and Distributed Caching." *IEEE/ACM Trans. Netw.*, (21)4:404-418, Aug. 2001.

ROSENBLUM, M. and GARFINKEL, T.: "Virtual Machine Monitors: Current Technology and Future Trends." *IEEE Computer*, (38)5:39-47, May 2005.

ROUSSOS, G., MARSH, A. J., and MAGLAVERA, S.: "Enabling Pervasive Computing with Smart Phones." *IEEE Pervasive Comput.*, (4)2:20-26, Apr. 2005.

ROWSTRON, A.: "Run-time Systems for Coordination." In Omicini, A., Zambonelli, F., Klusch, M., and Tolksdorf, R. (eds.), *Coordination of Internet Agents: Models, Technologies and Applications*, pp. 78-96. Berlin: Springer-Verlag, 2001.

ROWSTRON, A. and DRUSCHEL, P.: "Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems." *Proc. Middleware 2001*, vol. 2218 of *Lect. Notes Comp. Sc.* Berlin: Springer-Verlag, 2001. pp. 329-350.

ROWSTRON, A. and WRAY, S.: "A Run-Time System for WCL." In Bal, H., Belkhouche, B., and Cardelli, L. (eds.), *Internet Programming Languages*, vol. 1686 of *Lect. Notes Comp. Sc.*, pp. 78-96. Berlin: Springer-Verlag, 1998.

RUSSELLO, G., CHAUDRON, M., and VAN STEEN, M.: "Adapting Strategies for Distributing Data in Shared Data Space." *Proc. Int'l Symp. Distr. Objects & Appl. (DOA)*, vol. 3291 of *Lect. Notes Comp. Sc.*, (Agia Napa, Cyprus). Berlin: Springer-Verlag, 2004. pp. 1225-1242.

RUSSELLO, G., CHAUDRON, M., VAN STEEN, M., and BOKHAROUSS, I.: "Dynamically Adapting Tuple Replication for Managing Availability in a Shared Data Space." *Sc. Comp. Programming*, (63), 2006.

SADJADI, S. and MCKINLEY, P.: "A Survey of Adaptive Middleware." Technical Report MSU-CSE-03-35, Michigan State University, Computer Science and Engineering, Dec. 2003.

SAITO, Y. and SHAPIRO, M.: "Optimistic Replication." *ACM Comput. Surv.*, (37)1:42-81, Mar. 2005.

SALTZER, J. and SCHROEDER, M.: "The Protection of Information in Computer Systems." *Proceedings of the IEEE*, (63)9:1278-1308, Sept. 1975.

SALTZER, J.: "Naming and Binding Objects." In Bayer, R., Graham, R., and Seegmuller, G. (eds.), *Operating Systems: An Advanced Course*, vol. 60 of *Lect. Notes Comp. Sc.*, pp. 99-208. Berlin: Springer-Verlag, 1978.

SALTZER, J., REED, D., and CLARK, D.: "End-to-End Arguments in System Design." *ACM Trans. Comp. Syst.*, (2)4:277-288, Nov. 1984.

SANDHU, R. S., COYNE, E. J., FEINSTEIN, H. L., and YOUMAN, C. E.: "Role-Based Access Control Models." *IEEE Computer*, (29)2:38-47, Feb. 1996.

SAROIU, S., GUMMADI, P. K., and GRIBBLE, S. D.: "Measuring and Analyzing the

Characteristics of Napster and Gnutella Hosts." *ACM Multimedia Syst.*, (9)2:170-184, Aug. 2003. Cited on page 53.

SATYANARAYANAN, M.: "The Evolution of Coda." *ACM Trans. Comp. Syst.*, (20)2:85-124, May 2002.

SATYANARAYANAN, M. and SIEGEL, E.: "Parallel Communication in a Large Distributed System." *IEEE Trans. Comp.*, (39)3:328-348, Mar. 1990.

SAXENA, P. and RAI, J.: "A Survey of Permission-based Distributed Mutual Exclusion Algorithms." *Computer Standards and Interfaces*, (25)2:159-181, May 2003.

SCHMIDT, D., STAL, M., ROHNERT, H., and BUSCHMANN, F.: *Pattern-Oriented Software Architecture – Patterns for Concurrent and Networked Objects*. New York: John Wiley, 2000.

SCHNEIDER, F.: "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial." *ACM Comput. Surv.*, (22)4:299-320, Dec. 1990.

SCHNEIER, B.: *Applied Cryptography*. New York: John Wiley, 2nd ed., 1996.

SCHNEIER, B.: *Secrets and Lies*. New York: John Wiley, 2000.

SCHULZRINNE, H.: "The tel URI for Telephone Numbers." RFC 3966, Jan. 2005.

SCHULZRINNE, H., CASNER, S., FREDERICK, R., and JACOBSON, V.: "RTP: A Transport Protocol for Real-Time Applications." RFC 3550, July 2003.

SEBESTA, R.: *Programming the World Wide Web*. Reading, MA: Addison-Wesley, 3rd ed., 2006.

SHAPIRO, M., DICKMAN, P., and PLAINFOSSE, D.: "SSP Chains: Robust, Distributed References Supporting Acyclic Garbage Collection." Technical Report 1799, INRIA, Rocquencourt, France, Nov. 1992.

SHAW, M. and CLEMENTS, P.: "A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems." *Proc. 21st Int'l Comp. Softw. & Appl. Conf.*, 1997. pp. 6-13.

SHEPLER, S., CALLAGHAN, B., ROBINSON, D., THURLOW, R., BEAME, C., EISLER, M., and NOVECK, D.: "Network File System (NFS) Version 4 Protocol." RFC 3530, Apr. 2003.

SHETH, A. P. and LARSON, J. A.: "Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases." *ACM Comput. Surv.*, (22)3:183-236, Sept. 1990.

SHOOMAN, M. L.: *Reliability of Computer Systems and Networks: Fault Tolerance, Analysis, and Design*. New York: John Wiley, 2002.

SILBERSCHATZ, A., GALVIN, P., and GAGNE, G.: *Operating System Concepts*. New York: John Wiley, 7th ed., 2005.

SINGH, A., CASTRO, M., DRUSCHEL, P., and ROWSTRON, A.: "Defending Against Eclipse Attacks on Overlay Networks." *Proc. 11th SIGOPS European Workshop*, (Leuven, Belgium). New York, NY: ACM Press, 2004. pp. 115-120.

SINGH, A., NGAN, T.-W., DRUSCHEL, P., and WALLACH, D. S.: "Eclipse Attacks on Overlay Networks: Threats and Defenses." *Proc. 25th INFOCOM Conf.*, (Barcelona, Spain). Los Alamitos, CA: IEEE Computer Society Press, 2006.

SINGHAL, M. and SHIVARATRI, N.: *Advanced Concepts in Operating Systems: Distributed, Database, and Multiprocessor Operating Systems*. New York: McGraw-Hill, 1994.

SIVASUBRAMANIAN, S., PIERRE, G., and VAN STEEN, M.: "Replicating Web Applications On-Demand." *Proc. First Int'l Conf. Services Comput.*, (Shanghai, China). Los Alamitos, CA: IEEE Computer Society Press, 2004a. pp. 227-236.

SIVASUBRAMANIAN, S., PIERRE, G., VAN STEEN, M., and ALONSO, G.: "GlobeCBC: Content-blind Result Caching for Dynamic Web Applications." Technical Report, Vrije Universiteit, Department of Computer Science, Jan. 2006.

SIVASUBRAMANIAN, S., SZYMANIAK, M., PIERRE, G., and VAN STEEN, M.: "Replication for Web Hosting Systems." *ACM Comput. Surv.*, (36)3:1-44, Sept. 2004b.

SIVASUBRAMANIAN, S., ALONSO, G., PIERRE, G., and VAN STEEN, M.: "GlobeDB: Autonomic Data Replication for Web Applications." *Proc. 14th Int'l WWW Conf.*, (Chiba, Japan). New York, NY: ACM Press, 2005. pp. 33-42.

SIVRIKAYA, F. and YENER, B.: "Time Synchronization in Sensor Networks: A Survey." *IEEE Network*, (18)4:45-50, July 2004.

SKEEN, D.: "Nonblocking Commit Protocols." *Proc. SIGMOD Int'l Conf. on Management Of Data*. ACM, 1981. pp. 133-142.

SKEEN, D. and STONEBRAKER, M.: "A Formal Model of Crash Recovery in a Distributed System." *IEEE Trans. Softw. Eng.*, (SE-9)3:219-228, Mar. 1983.

SMITH, J. and NAIR, R.: "The Architecture of Virtual Machines." *IEEE Computer*, (38)5:32-38, May 2005.

SMITH, J. and NAIR, R.: *Virtual Machines: Versatile Platforms for Systems and Processes*. San Mateo, CA: Morgan Kaufman, 2005.

SNIR, M., OTTO, S., HUSS-LEDERMAN, S., WALKER, D., and DONGARRA, J.: *MPI: The Complete Reference – The MPI Core*. Cambridge, MA: MIT Press, 1998.

SPEAKMAN, T., CROWCROFT, J., GEMMELL, J., FARINACCI, D., LIN, S., LESHCHINER, D., LUBY, M., MONTGOMERY, T., RIZZO, L., TWEEDLY, A., BHASKAR, N., EDMONSTONE, R., SUMANASEKERA, R., and VICISANO, L.: "PGM Reliable Transport Protocol Specification." RFC 3208, Dec. 2001.

SPECHT, S. M. and LEE, R. B.: "Distributed Denial of Service: Taxonomies of Attacks, Tools, and Countermeasures." *Proc. Int'l Workshop on Security in Parallel and Distributed Systems*, (San Francisco, CA), 2004. pp. 543-550.

SPECTOR, A.: "Performing Remote Operations Efficiently on a Local Computer Network." *Commun. ACM*, (25)4:246-260, Apr. 1982.

SRINIVASAN, R.: "RPC: Remote Procedure Call Protocol Specification Version 2." RFC 1831, Aug. 1995a.

SRINIVASAN, R.: "XDR: External Data Representation Standard." RFC 1832, Aug. 1995b.

SRIPANIDKULCHAI, K., MAGGS, B., and ZHANG, H.: "Efficient Content Location Using Interest-Based Locality in Peer-to-Peer Systems." *Proc. 22nd INFOCOM Conf.*, (San Francisco, CA). Los Alamitos, CA: IEEE Computer Society Press, 2003.

STEIN, L.: *Web Security, A Step-by-Step Reference Guide.* Reading, MA: Addison-Wesley, 1998.

STEINDER, M. and SETHI, A.: "A Survey of Fault Localization Techniques in Computer Networks." *Sc. Comp. Programming*, (53)165-194, May 2004.

STEINER, J., NEUMAN, C., and SCHILLER, J.: "Kerberos: An Authentication Service for Open Network Systems." *Proc. Winter Techn. Conf.* USENIX, 1988. pp. 191-202.

STEINMETZ, R.: "Human Perception of Jitter and Media Synchronization." *IEEE J. Selected Areas Commun.*, (14)1:61-72, Jan. 1996.

STEINMETZ, R. and NAHRSTEDT, K.: *Multimedia Systems.* Berlin: Springer-Verlag, 2004.

STEVENS, W.: *UNIX Network Programming – Networking APIs: Sockets and XTI.* Englewood Cliffs, NJ: Prentice Hall, 2nd ed., 1998.

STEVENS, W.: *UNIX Network Programming – Interprocess Communication.* Englewood Cliffs, NJ: Prentice Hall, 2nd ed., 1999.

STEVENS, W. and RAGO, S.: *Advanced Programming in the UNIX Environment.* Reading, MA: Addison-Wesley, 2nd ed., 2005.

STOICA, I., MORRIS, R., LIBEN-NOWELL, D., KARGER, D. R., KAASHOEK, M. F., DABEK, F., and BALAKRISHNAN, H.: "Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications." *IEEE/ACM Trans. Netw.*, (11)1:17-32, Feb. 2003.

STOJMENOVIC, I.: "Position-based Routing in Ad Hoc Networks." *IEEE Commun. Mag.*, (40)7:128-134, July 2002.

STRAUSS, J., KATABI, D., and KAASHOEK, F.: "A Measurement Study of Available Bandwidth Estimation Tools." *Proc. Third Internet Measurement Conf.*, (Miami Beach, FL, USA). New York, NY: ACM Press, 2003. pp. 39–44.

SUGERMAN, J., VENKITACHALAM, G., and LIM, B.-H.: "Virtualizing I/O Devices on VMware Workstation s Hosted Virtual Machine Monitor." *Proc. USENIX Ann. Techn. Conf.*, (Boston, MA). Berkeley, CA: USENIX, 2001. pp. 1-14.

SUN MICROSYSTEMS: *Java Message Service, Version 1.1.* Sun Microsystems, Mountain Ciew, Calif., Apr. 2004a.

SUN MICROSYSTEMS: *Java Remote Method Invocation Specification, JDK 1.5.* Sun Microsystems, Mountain View, Calif., 2004b.

SUN MICROSYSTEMS: *EJB 3.0 Simplified API.* Sun Microsystems, Mountain View, Calif., Aug. 2005a.

SUN MICROSYSTEMS: *Jini Technology Starter Kit, Version 2.1*, Oct. 2005b.

SUNDARARAMAN, B., BUY, U., and KSHEMKALYANI, A. D.: "Clock Synchronization for Wireless Sensor Networks: A Survey." *Ad-Hoc Networks*, (3)3:281-323, May 2005.

SZYMANIAK, M., PIERRE, G., and VAN STEEN, M.: "Scalable Cooperative Latency Estimation." *Proc. Tenth Int'l Conf. Parallel and Distributed Systems*, (Newport Beach, CA). Los Alamitos, CA: IEEE Computer Society Press, 2004. pp. 367-376.

SZYMANIAK, M., PIERRE, G., and VAN STEEN, M.: "A Single-Homed Ad hoc Distributed Server." Technical Report IR-CS-013, Vrije Universiteit, Department of Computer Science, Mar. 2005.

SZYMANIAK, M., PIERRE, G., and VAN STEEN, M.: "Latency-driven replica placement." *IPSJ Digital Courier*, (2), 2006.

TAIANI, F., FABRE, J.-C., and KILLIJIAN, M.-O.: "A Multi-Level Meta-Object Protocol for Fault-Tolerance in Complex Architectures." *Proc. Int'l Conf. Dependable Systems and Networks*, (Yokohama, Japan). Los Alamitos, CA: IEEE Computer Society Press, 2005. pp. 270-279.

TAM, D., AZIMI, R., and JACOBSEN, H.-A.: "Building Content-Based Publish/Subscribe Systems with Distributed Hash Tables." *Proc. First Int'l Workshop on Databases, Information Systems and Peer-to-Peer Computing*, vol. 2944 of *Lect. Notes Comp. Sc.*, (Berlin, Germany). Berlin: Springer-Verlag, 2003. pp. 138-152.

TAN, S.-W., WATERS, G., and CRAWFORD, J.: "A Survey and Performance Evaluation of Scalable Tree-based Application Layer Multicast Protocols." Technical Report 9-03, University of Kent, UK, July 2003.

TANENBAUM, A.: *Computer Networks*. Upper Saddle River, NJ: Prentice Hall, 4th ed., 2003.

TANENBAUM, A., MULLENDER, S., and VAN RENESSE, R.: "Using Sparse Capabilities in a Distributed Operating System." *Proc. Sixth Int'l Conf. on Distributed Computing Systems*. IEEE, 1986. pp. 558-563.

TANENBAUM, A., VAN RENESSE, R., VAN STAVEREN, H., SHARP, G., MULLENDER, S., JANSEN, J., and VAN ROSSUM, G.: "Experiences with the Amoeba Distributed Operating System." *Commun. ACM*, (33)12:46-63, Dec. 1990.

TANENBAUM, A. and WOODHULL, A.: *Operating Systems, Design and Implementation*. Englewood Cliffs, NJ: Prentice Hall, 3rd ed., 2006.

TANISCH, P.: "Atomic Commit in Concurrent Computing." *IEEE Concurrency*, (8)4:34-41, Oct. 2000.

TARTALJA, I. and MILUTINOVIC, V.: "Classifying Software-Based Cache Coherence Solutions." *IEEE Softw.*, (14)3:90-101, May 1997.

TEL, G.: *Introduction to Distributed Algorithms*. Cambridge, UK: Cambridge University Press, 2nd ed., 2000.

TERRY, D., DEMERS, A., PETERSEN, K., SPREITZER, M., THEIMER, M., and WELSH, B.: "Session Guarantees for Weakly Consistent Replicated Data." *Proc. Third Int'l Conf. on Parallel and Distributed Information Systems*, (Austin, TX). Los Alamitos, CA: IEEE Computer Society Press, 1994. pp. 140-149.

TERRY, D., PETERSEN, K., SPREITZER, M., and THEIMER, M.: "The Case for Non-transparent Replication: Examples from Bayou." *IEEE Data Engineering*, (21)4:12-20, Dec. 1998.

THOMAS, R.: "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases." *ACM Trans. Database Syst.*, (4)2:180-209, June 1979.

TIBCO: *TIB/Rendezvous Concepts, Release 7.4*. TIBCO Software Inc., Palo Alto, CA, July 2005.

TOLIA, N., HARKES, J., KOZUCH, M., and SATYANARAYAN, M.: "Integrating Portable and Distributed Storage." *Proc. Third USENIX Conf. File and Storage Techn.*, (Boston, MA). Berkeley, CA: USENIX, 2004.

TOLKSDORF, R. and ROWSTRON, A.: "Evaluating Fault Tolerance Methods for Large-scale Linda-like systems." *Proc. Int'l Conf. on Parallel and Distributed Processing Techniques and Applications*, vol. 2, (Las Vegas, NV), 2000. pp. 793-800.

TOWSLEY, D., KUROSE, J., and PINGALI, S.: "A Comparison of Sender-Initiated and Receiver-Initiated Reliable Multicast Protocols." *IEEE J. Selected Areas Commun.*, (15)3:398-407, Apr. 1997.

TRIPATHI, A., KARNIK, N., VORA, M., AHMED, T., and SINGH, R.: "Mobile Agent Programming in Ajanta." *Proc. 19th Int'l Conf. on Distributed Computing Systems*. IEEE, 1999. pp. 190-197.

TUREK, J. and SHASHA, S.: "The Many Faces of Consensus in Distributed Systems." *IEEE Computer*, (25)6:8-17, June 1992.

UMAR, A.: *Object-Oriented Client/Server Internet Environments*. Upper Saddle River, NJ: Prentice Hall, 1997.

UPnP Forum: "UPnP Device Architecture Version 1.0.1, Dec. 2003.

VAN RENESSE, R., BIRMAN, K., and VOGELS, W.: "Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining." *ACM Trans. Comp. Syst.*, (21)2:164-206, May 2003.

VAN STEEN, M., HAUCK, F., HOMBURG, P., and TANENBAUM, A.: "Locating Objects in Wide-Area Systems." *IEEE Commun.*, (36)1:104-109, Jan. 1998.

VASUDEVAN, S., KUROSE, J. F., and TOWSLEY, D. F.: "Design and Analysis of a Leader Election Algorithm for Mobile Ad Hoc Networks." *Proc. 12th Int'l Conf. on Network Protocols*, (Berlin, Germany). Los Alamitos, CA: IEEE Computer Society Press, 2004. pp. 350-360.

VEIGA, L. and FERREIRA, P.: "Asynchronous Complete Distributed Garbage Collection." *Proc. 19th Int'l Parallel & Distributed Processing Symp.*, (Denver, CO). Los Alamitos, CA: IEEE Computer Society Press, 2005.

VELAZQUEZ, M.: "A Survey of Distributed Mutual Exclusion Algorithms." Technical Report CS-93-116, University of Colorado at Boulder, Sept. 1993.

VERISSIMO, P. and RODRIGUES, L.: *Distributed Systems for Systems Architects*. Dordrecht, The Netherlands: Kluwer Academic Publishers, 2001.

VETTER, R., SPELL, C., and WARD, C.: "Mosaic and the World-Wide Web." *IEEE Computer*, (27)10:49-57, Oct. 1994.

VITEK, J., BRYCE, C., and ORIOL, M.: "Coordinating Processes with Secure Spaces." *Sc. Comp. Programming*, (46)1-2, 2003.

VOGELS, W.: "Tracking Service Availability in Long Running Business Activities." *Proc. First Int'l Conf. Service Oriented Comput.*, vol. 2910 of *Lect. Notes Comp. Sc.*, (Trento, Italy). Berlin: Springer-Verlag, 2003. pp. 395-408.

VOULGARIS, S. and VAN STEEN, M.: "Epidemic-style Management of Semantic Overlays for Content-Based Searching." *Proc. 11th Int'l Conf. Parallel and Distributed Computing (Euro-Par)*, vol. 3648 of *Lect. Notes Comp. Sc.*, (Lisbon, Portugal). Berlin: Springer-Verlag, 2005. pp. 1143-1152.

VOULGARIS, S., RIVIERE, E., KERMARREC, A.-M., and VAN STEEN, M.: "Sub-2-Sub: Self-Organizing Content-Based Publish and Subscribe for Dynamic and Large Scale Col-

laborative Networks." *Proc. Fifth Int'l Workshop on Peer-to-Peer Systems*, (Santa Barbara, CA), 2006.

VOYDOCK, V. and KENT, S.: "Security Mechanisms in High-Level Network Protocols." *ACM Comput. Surv.*, (15)2:135-171, June 1983.

WAH, B. W., SU, X., and LIN, D.: "A Survey of Error-Concealment Schemes for Real-Time Audio and Video Transmissions over the Internet." *Proc. Int'l Symp. Multimedia Softw. Eng.*, (Taipei, Taiwan). Los Alamitos, CA: IEEE Computer Society Press, 2000. pp. 17-24.

WAHBE, R., LUCCO, S., ANDERSON, T., and GRAHAM, S.: "Efficient Software-based Fault Isolation." *Proc. 14th Symp. Operating System Principles*. ACM, 1993. pp. 203-216.

WALDO, J.: "Remote Procedure Calls and Java Remote Method Invocation." *IEEE Concurrency*, (6)3:5-7, July 1998.

WALFISH, M., BALAKRISHNAN, H., , and SHENKER, S.: "Untangling the Web from DNS." *Proc. First Symp. Networked Systems Design and Impl.*, (San Francisco, CA). Berkeley, CA: USENIX, 2004. pp. 225-238.

WALLACH, D.: "A Survey of Peer-to-Peer Security Issues." *Proc. Int'l Symp. Softw. Security*, vol. 2609 of *Lect. Notes Comp. Sc.*, (Tokyo, Japan). Berlin: Springer-Verlag, 2002. pp. 42-57.

WALLACH, D., BALFANZ, D., DEAN, D., and FELTEN, E.: "Extensible Security Architectures for Java." *Proc. 16th Symp. Operating System Principles*. ACM, 1997. pp. 116-128.

WANG, C., CARZANIGA, A., EVANS, D., and WOLF, A. L.: "Security Issues and Requirements for Internet-Scale Publish-Subscribe Systems." *Proc. 35th Hawaii Int'l Conf. System Sciences*, vol. 9. IEEE, 2002. pp. 303-310.

WANG, H., LO, M. K., and WANG, C.: "Consumer Privacy Concerns about Internet Marketing." *Commun. ACM*, (41)3:63-70, Mar. 1998.

WATTS, D. J.: *Small Worlds, The Dynamics of Networks between Order and Randomness*. Princeton, NJ: Princeton University Press, 1999.

WELLS, G., CHALMERS, A., and CLAYTON, P.: "Linda Implementations in Java for Concurrent Systems." *Conc. & Comput.: Prac. Exp.*, (16)10:1005-1022, Aug. 2004.

WESSELS, D.: *Squid: The Definitive Guide*. Sebastopol, CA: O'Reilly & Associates, 2004.

WHITE, S. R., HANSON, J. E., WHALLEY, I., CHESS, D. M., , and KEPHART, J. O.: "An Architectural Approach to Autonomic Computing." *Proc. First Int'l Conf. Autonomic Comput.*, (New York, NY). Los Alamitos, CA: IEEE Computer Society Press, 2004. pp. 2-9.

WIERINGA, R. and DE JONGE, W.: "Object Identifiers, Keys, and Surrogates–Object Identifiers Revisited." *Theory and Practice of Object Systems*, (1)2:101-114, 1995.

WIESMANN, M., PEDONE, F., SCHIPER, A., KEMME, B., and ALONSO, G.: "Understanding Replication in Databases and Distributed Systems." *Proc. 20th Int'l Conf. on Distributed Computing Systems*. IEEE, 2000. pp. 264-274.

WOLLRATH, A., RIGGS, R., and WALDO, J.: "A Distributed Object Model for the Java

System." *Computing Systems*, (9)4:265-290, Fall 1996.

WOLMAN, A., VOELKER, G., SHARMA, N., CARDWELL, N., KARLIN, A., and LEVY, H.: "On the Scale and Performance of Cooperative Web Proxy Caching." *Proc. 17th Symp. Operating System Principles*. ACM, 1999. pp. 16-31.

WU, D., HOU, Y., ZHU, W., ZHANG, Y., and PEHA, J.: "Streaming Video over the Internet: Approaches and Directions." *IEEE Trans. Circuits & Syst. Video Techn.*, (11)1:1-20, Feb. 2001.

YANG, B. and GARCIA-MOLINA, H.: "Designing a Super-Peer Network." *Proc. 19th Int'l Conf. Data Engineering*, (Bangalore, India). Los Alamitos, CA: IEEE Computer Society Press, 2003. pp. 49-60.

YANG, M., ZHANG, Z., LI, X., and DAI, Y.: "An Empirical Study of Free-Riding Behavior in the Maze P2P File-Sharing System." *Proc. Fourth Int'l Workshop on Peer-to-Peer Systems*, Lect. Notes Comp. Sc., (Ithaca, NY). Berlin: Springer-Verlag, 2005.

YELLIN, D.: "Competitive Algorithms for the Dynamic Selection of Component Implementations." *IBM Syst. J.*, (42)1:85-97, Jan. 2003.

YU, H. and VAHDAT, A.: "Efficient Numerical Error Bounding for Replicated Network Services." In Abbadi, Amr El, Brodie, Michael L., Chakravarthy, Sharma, Dayal, Umeshwar, Kamel, Nabil, Schlageter, Gunter, and Whang, Kyu-Young (eds.), *Proc.26th Int'l Conf. Very Large Data Bases*, (Cairo, Egypt). San Mateo, CA: Morgan Kaufman, 2000. pp. 123-133.

YU, H. and VAHDAT, A.: "Design and Evaluation of a Conit-Based Continuous Consistency Model for Replicated Services." *ACM Trans. Comp. Syst.*, (20)3:239–282, 2002.

ZHANG, C. and JACOBSEN, H.-A.: "Resolving Feature Convolution in Middleware Systems." *Proc. 19th OOPSLA*, (Vancouver, Canada). New York, NY: ACM Press, 2004. pp. 188-205.

ZHAO, B., HUANG, L., STRIBLING, J., RHEA, S., JOSEPH, A., and KUBIATOWICZ, J.: "Tapestry: A Resilient Global-Scale Overlay for Service Deployment." *IEEE J. Selected Areas Commun.*, (22)1:41-53, Jan. 2004.

ZHAO, F. and GUIBAS, L.: *Wireless Sensor Networks*. San Mateo, CA: Morgan Kaufman, 2004.

ZHAO, Y., STURMAN, D., and BHOLA, S.: "Subscription Propagation in Highly-Available Publish/Subscribe Middleware." *Proc. Middleware 2004*, vol. 3231 of *Lect. Notes Comp. Sc.*, (Toronto, Canada). Berlin: Springer-Verlag, 2004. pp. 274-293.

ZHU, Q., CHEN, Z., TAN, L., ZHOU, Y., KEETON, K., and WILKES, J.: "Hibernator: Helping Disk Arrays Sleep through the Winter." *Proc. 20th Symp. Operating System Prin., (Brighton, UK). New York, NY: ACM Press, 2005. pp. 177-190.*

ZHUANG, S. Q., GEELS, D., STOICA, I., and KATZ, R. H.: "On Failure Detection Algorithms in Overlay Networks." *Proc. 24th INFOCOM Conf.*, (Miami, FL). Los Alamitos, CA: IEEE Computer Society Press, 2005.

ZOGG, J.-M.: "GPS Basics." Technical Report GPS-X-02007, UBlox, Mar. 2002.

ZWICKY, E., COOPER, S., CHAPMAN, D., and RUSSELL, D.: *Building Internet Firewalls*. Sebastopol, CA: O'Reilly & Associates, 2nd ed., 2000.

DISTRIBUTED SYSTEMS PRINCIPLES AND PARADIGMS

分布式系统原理与范型（第2版）

本书是著名作者Tanenbaum关于分布式系统的最新力作，是分布式系统的权威教材。本书分成了两大部分。第2～9章讨论的是分布式系统的的原理、概念和技术，包括通信、进程、命名、同步化、一致性和复制、容错性以及安全性等，而分布式应用的开发方法（即范型）在第10～13章中进行了讨论。但是，与前一版不同的是，我们没有在讨论范型的章节中完整地介绍每个案例研究，而是通过一个有代表性的案例来解释原理。这种介绍方法使得我们不仅精简了素材，而且可以使得读者在阅读和学习时更愉快些。全书结构清晰，内容全面经典，系统性与先进性并茂。

本书适用对象广泛，不仅可以作为学习分布式计算机的本科生和研究生的教材，对于从事分布式计算研究和工程应用的科技人员和工程技术人员来说，本书也是一本优秀的读物。

世界著名计算机教材精选

- 计算机网络（第4版）
  Computer Networks, 4E
- TCP/IP协议族（第3版）
  TCP/IP Suite, 3E
- 软件体系结构
  Software architecture
- 算法设计
  Algorithm Design
- 标准C程序设计（第3版）
  Programming in ANSI C, 3E
- Visual Basic 2005程序设计（第6版）
  An Introduction to Programming Using Visual Basic 2005, 6E
- 计算机组织与体系结构：性能设计（第7版）
  Computer Organization and Architecture, 7E
- 逻辑设计基础（第2版）
  Introduction to Logic Design, 2E
- 数据结构与算法分析 ——C++语言描述（第2版）
  ADTs, Data Structures, and Problem Solving with C++, 2E
- 面向对象软件工程：使用UML、模式与Java（第2版）
  Object-Oriented Software Engineering, 2E
- Java面向对象程序设计（第2版）
  Introduction to Programming using Java, 2E
- 计算理论基础（第2版）
  Elements of The Theory of Computation, 2E
- 程序设计语言概念
  Programming Language Concepts
- 3D计算机图形学（OpenGL版）
  3D Computer Graphics
- 无线通信与网络（第2版）
  Wireless Communications and Networks, 2E
- 操作系统原理
  Operating Systems Principles
- 数据通信——原理、技术与应用（第5版）
  Business Data Communications, 5E
- 密码学与网络安全
  Cryptography and Network Security
- 算法基础
  Fundamentals of Algorithmics
- Java程序设计
  Java Program Design
- 经典密码学与现代密码学
  Classical and Contemporary Cryptology
- 数据挖掘教程
  Data Mining: Introductory and Advanced Topics
- 面向对象设计UML实践(第2版)
  Practical Object-Oriented Design with UML, 2E
- 分布式系统原理与范型(第2版)
  Distributed Systems: Principles and Paradigms, 2E
- 数据结构与算法分析 ——Java语言描述(第2版)
  Data Structures and Abstractions with Java, 2E
- 数据库管理系统原理与设计（第3版）
  Database Management Systems, 3E
- 计算机组成和设计硬件／软件接口（第2版）
  Computer Organization and Design, 2E
- 面向对象系统分析与设计
  Object-Oriented Systems Analysis and Design
- 数据库系统基础教程
  A First Course in Database Systems

ISBN 978-7-302-17279-6

9 787302 172796

PEARSON Prentice Hall

http://www.pearsoned.com

定价：59.00元